1-28-2021

Automated Test Generation for Validating SystemC Designs

Bin Lin
Portland State University

Automated Test Generation for Validating SystemC Designs

by

Bin Lin

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Fei Xie, Chair
Jingke Li
Suresh Singh
Xiaoyu Song

Portland State University
2021

ABSTRACT

Modern system design involves integration of all components of a system on a single chip, namely System-on-a-Chip (SoC). The ever-increasing complexity of SoCs and rapidly decreasing time-to-market have pushed the design abstraction to the electronic system level (ESL), in order to increase design productivity. SystemC is a widely used ESL modeling language that plays a central role in modern SoCs design process. ESL SystemC designs usually serve as executable specifications for the subsequent SoCs design flow. Therefore, undetected bugs in ESL SystemC designs may propagate to low-level implementations or even final silicon products. In addition, modern SoCs design often involves intellectual properties supplied by outsourced design services and untrusted third-party vendors, as well as intensive usage of electronic design automation tools provided by different vendors. Given this situation, modern SoCs are vulnerable to malicious implants such as hardware Trojans. Bugs and Trojans in silicon products can be extremely expensive and dangerous, especially in safety critical systems. Therefore, it is critical to detect bugs and Trojans as early as possible during SoCs design process. However, it is a challenging task for SystemC designs due to their object-oriented features and inherent concurrency, as well as the stealthy nature of hardware Trojans.

We propose a framework to validate SystemC designs with automated test generation. We first develop an approach for generating high-quality test cases

for SystemC designs using symbolic execution. To improve the scalability, we further propose an approach to test generation for SystemC designs with binary-level concolic testing techniques. To evaluate the quality of the generated test cases, we adopt code coverage and assertion-based verification techniques. We further extend our test generation framework for hardware Trojan detection in behavioral SystemC designs.

In addition, we also develop a comprehensive suite of benchmark designs for SystemC verification and validation. SystemC verification has been studied for around two decades. However, so far, different verification approaches are evaluated on different sets of SystemC designs, among which some designs are not updated according to the latest SystemC Standard. Lacking common benchmarks makes it difficult to compare the performances of various approaches. Our benchmark covers many application domains and SystemC core features, as well as conforming to the latest SystemC Standard.

To evaluate the efficiency, effectiveness and scalability of our test generation framework, we have applied it to the benchmark that we developed. Our experimental results demonstrate that the test cases generated by our approaches are able to achieve high code coverage and detect design errors effectively. In our experiments, our framework detects two severe errors, one functional error and one out-of-bound access. We have also applied our hardware Trojan detection approach to an open source SystemC benchmark with various hardware Trojans. Our approach is able to detect those hardware Trojans effectively and efficiently. The extensive experiments with our framework show that it scales to designs with practical sizes.

DEDICATION

In loving memory of my grandfather, Shaorong

To my parents, Qianlu and Aizhi

To my older sister, Lihong

To my wife, Ru

To my son, Luke

ACKNOWLEDGMENTS

Without the generous help and support from many professors, staffs, and colleagues in the department of computer science, as well as many friends and my family, this dissertation could not have been accomplished. I would like to express my sincere appreciation to all of them.

First and foremost, I would like to express my gratitude to Prof. Xie. I have been very fortunate to have Prof. Xie as my advisor. I am deeply grateful for his strong support and enlightening guidance through my Ph.D. study. He taught me how to conduct research and become an independent researcher. Without his continuous support and insightful advice, this dissertation would not have been accomplished. His solid knowledge and professional expertise with positive attitude have influence on my Ph.D. study and future career deeply.

I would like to thank Prof. Jingke Li, Prof. Suresh Singh, and Prof. Xiaoyu Song for serving on my doctoral committee. Thanks for their inspirational feedback for my research and valuable comments on my dissertation. I am grateful for their sacrifice of valuable time.

Thanks to Dr. Kai Cong and Dr. Zhenkun Yang, many research ideas of this dissertation come from productive discussions with them. I sincerely thank them for being excellent collaborators. I am also grateful that I have this great opportunity to work with a talented group: Christopher Havlicek, Jinchao Chen,

Dejun Qian, Li Lei, Jialu Wang, Qin Wang, Haifeng Gu, Zhe Li, and Lai Xu.

I would also like to thank my parents, older sister and brother-in-law for their continuous support and endless love. I could never have accomplished this without their unconditional support. I would express my special thanks to my wife Ru Jia for her sound and complete love. In the past six years, she shared every moment with me and encouraged me when I felt frustrated. Last but not least, I would like to recognize the influence of my little son, Luke, who was born during this dissertation writing. He has been bringing me so much joy and hope.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

SoC	System on a Chip
TLM	Transaction-Level Modeling
ESL	Electronic System Level
RTL	Register Transfer Level
IP	Intellectual Property
EDA	Electronic Design Automation
DUV	Design Under Validation
ABV	Assertion-Based Verification
FIFO	First In, First Out
HLS	High-Level Synthesis
IR	Intermediate Representation
LoC	Line of Code
SAT	Boolean Satisfiability Problem
SMT	Satisfiability Modulo Theories
SMV	Symbolic Model Verifier
SMC	Symbolic Model Checking
BMC	Bounded Model Checking
POR	Partial Order Reduction

Chapter 1

INTRODUCTION

1.1 MOTIVATION AND PROBLEM STATEMENT

1.1.1 Motivation

Modern system design involves integration of all components of a system on a single chip, namely System-on-a-Chip (SoC). The ever-increasing complexity of SoCs and rapidly decreasing time-to-market have pushed the design abstraction to the electronic system level (ESL), in order to increase design productivity. SystemC [49] is a widely adopted ESL modeling language that plays a central role in modern SoCs design process. To enable early exploration of design spaces and verification at a higher level of abstraction, SystemC has been widely used for system-level modeling, architectural exploration, functional verification, and high-level synthesis. These SystemC designs serve as executable specifications for the subsequent SoCs design flow. Therefore, quality assurance of SoCs in ESL is extremely important, since undetected bugs in these designs may propagate to low-level implementations or even final silicon products. Cost of detecting and fixing bugs in low-level implementations is much higher than in ESL. Moreover, bugs that remain undiscovered in final silicon products can be extremely expensive. If buggy products are released to the market, they may cause catastrophic consequences and even endanger lives,

especially in safety-critical systems. On the other hand, quality assurance of SoCs developed with the SystemC language is very challenging. First, the SystemC language heavily uses objected-oriented features, hardware-oriented data types, and inherent concurrency. Second, modern SoCs design often involves intellectual properties (IPs) supplied by outsourced design services and untrusted third-party vendors, as well as intensive usage of electronic design automation (EDA) tools provided by various vendors. Given this situation, modern SoCs are vulnerable to malicious implants such as hardware Trojans that intend to add, delete, or modify functionalities of SoCs. Hardware Trojans are stealthy in nature because they are only triggered under very rare conditions. Therefore, innovative approaches are highly demanded to find bugs and Trojans in ESL SystemC designs.

Existing formal approaches to SystemC verification is not scalable yet, because formal methods require formal semantics that describe the transition relation of a design. This is nontrivial for SystemC designs due to their heavy usage of object-oriented features, event-driven simulation semantics, and inherent concurrency. Dynamic validation, also known as the simulation-based approach, is the workhorse of SystemC validation [96]. SystemC simulation requires a set of concrete test cases. However, test cases for SystemC simulation are manually written or randomly generated so far. Manual test writing requires indepth knowledge of a design under validation (DUV), which is time-consuming, labor-intensive, and error-prone. Random testing, in contrast, is fast. However, many redundant test cases may be generated, which results in long simulation time. Furthermore, random testing usually leaves hard-to-reach segments and corner cases unexplored where bugs are likely to appear and hardware Trojans are often hidden.

Recently, symbolic execution [54], which can generate effective test cases and

achieve high code coverage, has been widely used for test generation [8, 13, 26, 67]. To mitigate the path explosion problem of pure symbolic execution, concolic (a portmanteau of concrete and symbolic) testing [90] that combines concrete execution and symbolic execution has achieved considerable success in both software and hardware domains [25, 34, 37, 91]. Symbolic execution and concolic testing are good at reaching corner cases and find deep bugs. Thus, they have great potential to play an important role in generating high-quality test cases that are able to detect design errors and hardware Trojans in SystemC designs.

1.1.2 Problem statement

This dissertation research is concerned with automated test generation and bug detection, as well as hardware Trojan detection in SystemC designs during pre-silicon stage. We observe the following three key challenges to achieve our goals.

- *High complexity of SystemC designs.* The SystemC language heavily utilizes object-oriented features and hardware-oriented data structures, as well as event-driven simulation semantics and inherent concurrency. These characteristics make it very challenging to generate high-quality test cases that are able to achieve high code coverage, and detect design errors and hardware Trojans in SystemC designs.

- *High stealthiness of hardware Trojans.* Hardware Trojans are embedded into a design with malicious intent such as functionality modification, sensitive information leakage, and denial of services. In order to not be discovered during design validation phase, hardware Trojans are usually only triggered under rare conditions. This stealthy nature makes them very hard to be detected during functional validation process.

- *Lacking of common and updated benchmarks.* In the literature, different SystemC verification approaches are evaluated on different sets of SystemC designs, among which some designs are not updated according to the latest SystemC Standard. Lacking of common and updated benchmarks makes it difficult to evaluate the performances among various verification approaches.

1.2 PROPOSED SOLUTION

We have proposed a scalable framework that generates test cases automatically for SystemC designs in the early SoCs design stage. Test cases generated by our framework are able to achieve high code coverage, and detect design errors and hardware Trojans effectively. In particular, this dissertation includes the following three key components.

- *Test generation and bug detection.* High-quality test cases that are able to achieve high code coverage and detect bugs are critical to simulation-based validation of SystemC designs. However, test cases are manually written or randomly generated so far, which is not effective. We have proposed and developed high-quality test case generation approaches for SystemC designs.

- *Hardware Trojan detection.* New design methodologies such as outsourced design services and widely used third-party IPs result in the partial relinquishment of the control over SoCs design process. Therefore, hardware vulnerabilities such as hardware Trojans have raised serious concerns. On the other hand, it is very challenging to detect hardware Trojans because they are only triggered in very rare conditions. We have developed an approach that is able to detect hardware Trojans in behavioral SystemC designs effectively and efficiently.

- *A suite of benchmark designs.* Common SystemC benchmarks, which are not available currently, are required to evaluate the performances of SystemC verification and validation approaches. We aimed to fill this gap by developing a benchmark that is freely available online. Therefore, we have developed SCBench, a comprehensive suite of benchmark designs for SystemC verification and validation. The benchmark covers a variety of application domains and most core features of the SystemC language.

More details of these components are summarized in the following.

Symbolic Execution of SystemC Designs. We first developed an approach to test generation for SystemC designs using symbolic execution. It includes three key steps: test-harness generation, symbolic execution, and test-case generation. A SystemC DUV and its test harness are compiled together to LLVM bitcode. Then, the symbolic execution engine takes the LLVM bitcode as input and executes it symbolically. For each explored path, the path constraints that are represented by symbolic expressions are sent to a constraint solver, which returns a test case if the constraints are satisfiable.

Concolic Testing of SystemC Designs. Although symbolic execution of SystemC designs is able to generate high code coverage test cases, it requires much manual effort to model hardware-oriented data types. Therefore, we developed an approach to generating test cases for SystemC designs in binary-level using concolic testing techniques. First, a given SystemC DUV is compiled into an executable binary by linking the SystemC library. Then, the binary is executed in a virtual machine concretely, during which the concrete execution trace is dumped. Afterwards, the dumped trace is explored by a symbolic execution engine to generate test cases. In addition, we have integrated assertion-based verification (ABV)

techniques to detect design errors.

Hardware Trojan Detection in SystemC Designs. Symbolic execution and concolic testing are good at generating test cases that are able to reach corner cases where hardware Trojans are usually hidden. Thus, we developed an approach to detecting hardware Trojans in behavioral SystemC designs with concolic testing. We proposed two optimizations, namely selective concolic testing and coverage-guided state search strategy, to improve the efficiency of traditional concolic testing.

SCBench Benchmark. The benchmark consists of 38 well-written representative SystemC designs that cover a variety of application domains such as CPU architecture, security, digital signal processing (DSP), networking, and artificial intelligence (AI). The designs range from small single process designs to large multi-process designs. All designs are selected carefully to cover as many SystemC core features as possible. Each design has been provided a set of stimuli and a testbench that includes stimuli applications and output monitors. Most importantly, SCBench is freely available online to all researchers [85].

1.3 DISSERTATION OUTLINE

This dissertation is organized as follows. Chapter 2 introduces the necessary background to understand this dissertation better including the SystemC language, symbolic execution, concolic testing, hardware Trojans, and the definition of a test case. Chapter 3 presents high coverage test-case generation for SystemC designs using symbolic execution. Chapter 4 describes our improved approach for validating SystemC designs through concolic testing techniques with integration of ABV techniques. Chapter 5 presents hardware Trojan detection in behavioral SystemC

designs with selective concolic testing. Chapter 6 provides SCBench, a comprehensive suite of benchmark designs for SystemC verification and validation. Chapter 7 talks about related work. Chapter 8 concludes this dissertation and discusses future research directions based on this dissertation.

Chapter 2

BACKGROUND

This chapter first introduces the background on SystemC including its language, design methodology with SystemC, and verification of SystemC designs. Then, we present the techniques, specifically symbolic execution and concolic testing, which are used in this dissertation research. We also introduce necessary background on hardware Trojans to enable better understanding this dissertation. In the end, we provide the definition of a test case for a SystemC design.

2.1 SYSTEMC

2.1.1 SystemC Language

SystemC [49] is a hardware description language based on C++. It heavily uses object-oriented features, such as templates, virtual functions, and inheritance. SystemC adopts a layered approach for the flexibility of introducing new constructs. The bottommost layer is the standard C++, on top of which, it is the *core language* that consists of an event-driven simulation kernel and backbone elements, such as modules, processes, ports, channels, events, and interfaces. Modules are the basic building blocks in SystemC. A module is a container that consists of at least one process to describe certain functionalities of a system. There are three types of processes: method process `SC_METHOD`, thread process `SC_THREAD`, and clocked thread process `SC_CTHREAD`. A module can also contain other modules to represent

the hierarchy of a system. Modules communicate through ports that are connected by channels.

A system modeled in SystemC can be implemented in software, hardware, or a combination of the two. To model hardware designs, the standard built-in C++ data types are lacking. To this end, SystemC provides a rich collection of hardware-oriented data types, such as bit vectors, fixed-point types, fixed-precision and arbitrary-precision integral types.

Processes of a SystemC design run concurrently. SystemC has event-driven simulation semantics and cooperative multitasking scheduling mechanisms. Each process is executed without interruption up to completion or to a predefined yield point such as function `wait()`. The detailed scheduling algorithm includes five phases as demonstrated in Figure 2.1. Each phase is described in the following.

(1) **Initialization phase**. Add every method and thread process to the runnable set, but exclude those processes that have called `dont_initialize()` and clocked thread processes.

(2) **Evaluation phase**. Select a process, remove it from the runnable set, and then trigger or resume its execution. Repeat this step until the runnable set is empty; then go to step (3).

(3) **Update phase**. Execute all pending calls to `update()` from calls to `request_update()` made in step (2).

(4) **Delta notification phase**. If there are pending delta notifications, determine which processes are sensitive to them and add those processes to the runnable set. Go to step (2) if the runnable set is non-empty. Otherwise, go to step (5).

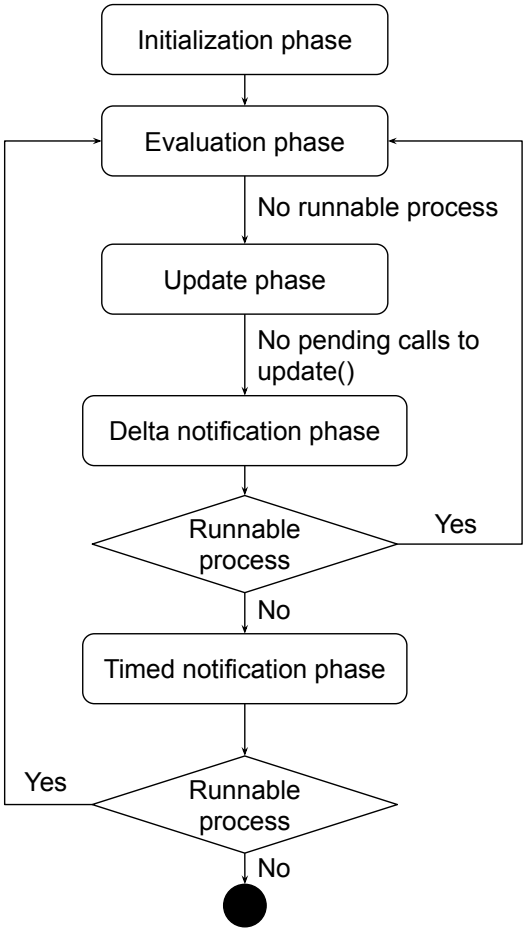

Figure 2.1: Semantics of the SystemC scheduler

(5) **Timed notification phase**. If there are pending timed notifications, advance simulation time to the earliest pending timed notification, determine which processes are sensitive to the events notified at the current time, and add those processes to the runnable set. Go to step (2) if the runnable set is non-empty. Otherwise, the simulation is finished.

SystemC is also widely used for transaction-level modeling (TLM), which is a high-level approach to ESL designs with focus on communication between processes rather than algorithms actually performed by processes. Thus, on top of the SystemC class library, there is a TLM-2.0 class library, which is particularly focused on memory-mapped bus modeling. The TLM-2.0 classes include core interfaces, sockets, generic payload, base protocol, as well as multiple utilities.

Figure 2.2 shows an example of a simple SystemC design. It has two processes P1 and P2. The inputs of the design are *en, clk, din*. The output is *dout*. The signal *data* connects P1 and P2. The variable *c* is local to process P2. Both processes are registered as clocked thread process SC_CTHREAD and sensitive to the positive edge of the clock. Each process will be executed at the positive edge of each clock cycle. The processes are suspended when they encounter function `wait()` that waits for the positive edge of next clock.

2.1.2 Design Methodology with SystemC

Figure 2.3 shows the SystemC based design methodology. SystemC can be used for hardware and software codesign and co-verification. It usually starts with a functional model on the system or architectural level. After the system level validation, the system is partitioned into a hardware part and a software part. The hardware part starts at the SystemC TLM, which serves as an executable platform

```
1  SC_MODULE(example) {
2    // input ports
3    sc_in<bool> en;
4    sc_in<bool> clk;
5    sc_in<int> din;
6
7    // output ports
8    sc_out<int> dout;
9
10   sc_signal<int> data;
11
12   void P1() {
13     wait();
14
15     while (true) {
16       if (en.read()) {
17         data = din.read();
18       }
19       wait();
20     }
21   }
20   void P2() {
21     wait();
22     while (true) {
23       int c = data.read();
24       if (c < 0) {
25         dout.write(-1 * c);
26       } else if (c % 2) {
27         dout.write(1);
28       } else {
29         dout.write(0);
30       }
31
32       wait();
33     }
34   }
35
36   SC_CTOR(example) {
37     SC_CTHREAD(P1, clk.pos());
38     SC_CTHREAD(P2, clk.pos());
39   }
40 };
```

Figure 2.2: An example of a SystemC design

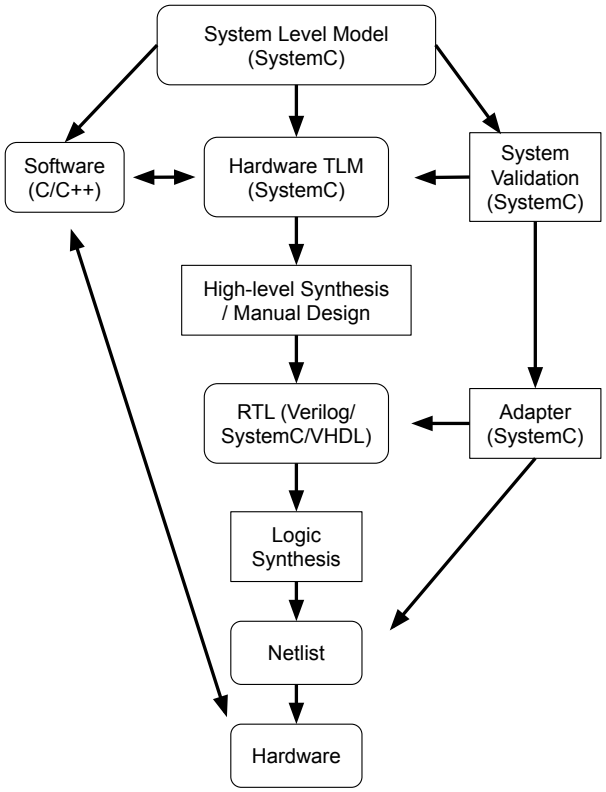

Figure 2.3: SystemC design methodology

and a golden reference model after validation. The SystemC TLM executable platform is accurate enough to execute software on, which enables earlier software and system validation. Then, the SystemC TLM is translated into RTL by high-level synthesis tools automatically or hardware engineers manually. The resultant RTL design can also be in the SystemC language. With this design flow, the same SystemC language can be used in multiple abstraction levels, such as system level functional model, hardware TLM, and RTL model. Thus, the high level validation effort such as testbench and test cases can be used directly or adapted slightly to validate the later low level models.

There are three primary advantages with this SystemC design methodology. First, it substantially increases simulation performance at the SystemC TLM level

over simulation platforms modeled at the RTL with Verilog or VHDL. Second, SystemC allows multiple abstraction levels, from system-level functional model down to cycle-accurate RTL implementation. This bridges the disconnect between system level model and RTL implementation within the traditional design methodology. This also enables verification reuse in that testbench and test cases developed for high-level designs may be reused for the later low-level implementations directly or by slightly adaptation, as shown in Figure 2.3. Third, the SystemC TLM executable platform is not only used for hardware verification, but also used for software development and system validation. This enables earlier verification during an SoC design process, consequently shortening the overall development time and decreasing the time-to-market.

2.1.3 SystemC Verification

The goal of verification is to discover errors or bugs in design models as early as possible, since the cost to fix a bug increases dramatically along the design stages. An industry study [33] has shown that verification accounts for 55% total project time on average from 2012 to 2016. In addition, the mean peak number of verification engineers is greater than the mean peak number of design engineers, and the number of verification engineers increases faster than the number of design engineers. This indicates that verification accounts for more than half of total project budget including both time cost and human effort. Furthermore, only 30% of design projects achieve first spin success. Re-spins not only postpone time-to-market of products, but also increase total costs including both money and human effort.

SystemC is widely used in the semiconductor industry to model a system at

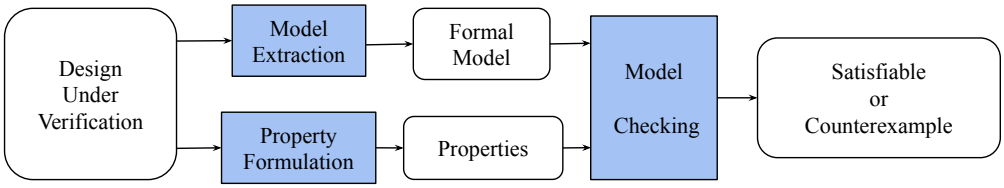

Figure 2.4: Verification flow with model checking

multiple abstraction levels, which enables stepwise refinement of a system-level design down to a low-level implementation. However, design errors in the system-level designs may propagate to low-level implementations. Detecting and fixing errors in these SystemC designs, namely SystemC verification, is very important and necessary. SystemC verification is meant to assure that SystemC designs implement the specifications correctly, which demands innovative approaches.

The existing approaches to SystemC verification can be largely divided into two paradigms: *formal verification* and *simulation-based verification*, also known as dynamic validation. Formal verification uses mathematical techniques to prove the correctness of a design. Three widely used formal methods are model checking [23], equivalence checking [57], and theorem proving [73]. However, model checking has been particularly used for SystemC verification in the literature. Model checking first abstracts a formal model from a DUV and formulates properties to be checked in the form of temporal logic. The idea is to determine whether a property holds by exploring the reachable states of a DUV exhaustively. The workflow of model checking is shown in Figure 2.4. If a property does not hold, the model checker will generate a counterexample that demonstrates the violation of the property.

Explicit and symbolic representations of states are two primary methods for model checking. Explicit-state model checking indexes states directly and explores

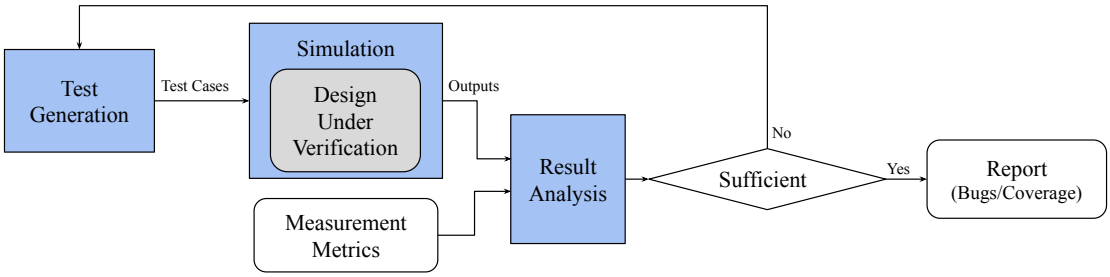

Figure 2.5: Verification flow with simulation-based approaches

state space of a design using graph algorithms starting from an initial state. Symbolic model checking (SMC) represents states with boolean encoding by which it can handle many more states than explicit methods. To overcome the state-space explosion problem, partial order reduction (POR) [35] and bounded model checking (BMC) [22] are two widely used techniques to alleviate the problem.

Simulation-based verification simulates a SystemC design with a set of concrete test cases and compare the simulation results with their expected outputs to discover design errors. The typical flow of simulation-based validation approaches is shown in Figure 2.5. The validation flow consists of three key steps: test generation, simulation, and result analysis. Generating high-quality test cases is a key challenge for simulation-based approaches. Traditionally, directed testing and random testing using constrained and unconstrained random approaches are both widely used in industrial practice. Directed testing typically requires indepth knowledge of a DUV and involves much human effort. Therefore, it is time-consuming, labor-intensive, and error-prone. Random testing, in contrast, is fast. However, many redundant test cases may be generated, which results in low performance and long simulation time. Moreover, random testing usually leaves hard-to-reach segments and corner cases unexplored, where bugs are likely to appear. Thus, additional innovative approaches are desired to generate effective test

cases automatically. In this work, we propose a framework to generate test cases automatically for SystemC designs using symbolic execution and concolic testing techniques.

2.2 SYMBOLIC EXECUTION

Symbolic execution [54] explores a program by taking symbolic values as inputs, which are symbols representing arbitrary values allowed by the types of corresponding variables. Consequently, the outputs of the program are represented as a function of its symbolic inputs. A symbolic execution state includes values of program variables, a path condition, and a program counter. The values of program variables can be concrete values or symbolic expressions over the symbolic inputs. The path condition collects constraints that must be satisfied to reach current execution state from the initial state. The constraints are represented as symbolic expressions over the symbolic inputs. The program counter denotes the next statement to execute.

We use the main body (*while* loop) of process P2 in Figure 2.2 to demonstrate how symbolic execution is performed, as shown in Figure 2.6. Suppose the value got from *data* is symbolic, then variable c becomes a symbolic variable that an arbitrary value of integer type at the point. When symbolic execution is performed, the path constraints are collected and updated at each branch point. There are two possible execution paths for each branch point whose condition depends on symbolic variables. For example at line 2, two symbolic execution states will be generated, one with constraint $c < 0$ and the other with constraint $c >= 0$. Similarly, two symbolic execution states will be generated at line 4, one with constraints $(c >= 0) \wedge (c \% 2 == 1)$ and the other with constraints $(c >= 0)$

```
1  int c = data.read();
2  if (c < 0) {
3    dout.write(−1 ∗ c);
4  } else if (c % 2) {
5    dout.write(1);
6  } else {
7    dout.write(0);
8  }
9  wait();
```

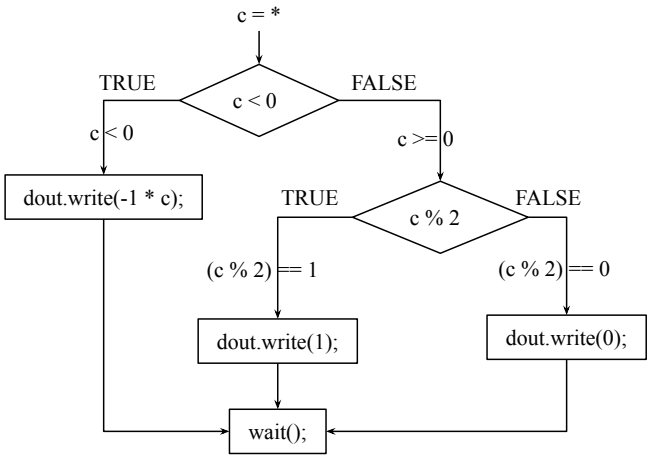

Figure 2.6: A symbolic execution example

$\wedge$ *(c % 2 == 0)*. One path is executed at a time and all possible paths will be explored eventually. For this example, three paths in total will be explored based on symbolic execution.

2.3 CONCOLIC TESTING

Concolic testing [90] is a hybrid verification technique that uses concolic execution, which partly addresses the limitations of random testing and symbolic execution. Concolic execution combines concrete execution and symbolic execution by making input values symbolic in addition to concrete. The concrete execution part performs normal execution of a program, while the symbolic execution part collects path constraints over the symbolic inputs at each branch point along the concrete execution path. The collected constraints are negated one condition at a time and sent to a constraint solver. If the solver can solve the negated constraints successfully, new test cases will be generated.

Figure 2.7 shows the process of concolic test generation. Solid red arrows denote

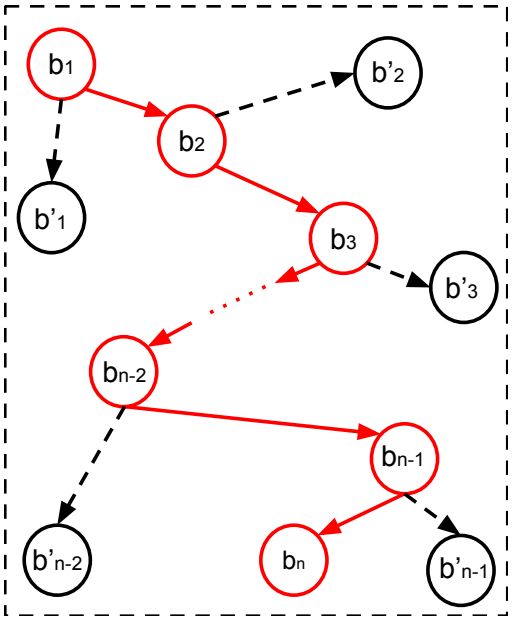

Figure 2.7: Concolic Test Generation

a concrete execution path, while dashed black arrows represent possible alternative paths where new test cases may be generated. During a concrete execution, symbolic constraints along the concrete execution path are collected. At each branch point, the constraints are negated and then solved if possible to generate a new test case which would lead the program along an alternative path.

2.4 HARDWARE TROJAN

Until recently, most of computer security research was devoted to software security. The underlying hardware was expected to be secure. However, this is no longer the case with the continuous globalization of the design and fabrication of modern SoCs, as well as the emergence of new design paradigms such as outsourced design services. Hardware Trojan has gained attention recently due to the emergence of its attacks. Hardware Trojan can be defined as any addition or modification to an electronic circuit or design with malicious intention including functionality

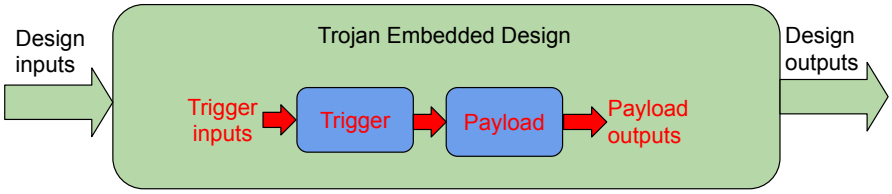

Figure 2.8: Generic structure of a hardware Trojan in a design

modification, sensitive information leakage, or denial of services. Hardware Trojan may be classified into several categories based on various characteristics, such as abstraction level, activation mechanism, and action type. A detailed taxonomy of hardware Trojans can be found in the survey paper [4]. In general, a hardware Trojan includes two parts, *trigger* (the activation mechanism) and *payload* (the action or the damage that a hardware Trojan will do once it is activated), as demonstrated by Figure 2.8. Hardware Trojans are usually stealthy and are triggered under rare conditions so that they are hard to be detected during functional validation.

2.5 PRELIMINARY DEFINITIONS

In order to help better understand this dissertation, we introduce the definition of a test case for a SystemC design.

Definition 2.1 (Input Port)**.** An input port p is a port which takes the input from a testbench. The `read()` function of p is called to get the input value.

As the design shown in Figure 2.2, variable *din* represents a SystemC input port while *din*.`read()` is called to get the input value. For each clock cycle, *din*.`read()` can get different inputs from the testbench.

Definition 2.2 (Input Port Set)**.** An input port set P is a set of input ports of a SystemC design.

As the design shown in Figure 2.2, the design input port set P includes three input ports *en*, *clk*, and *din*.

Definition 2.3 (Cycle Input)**.** A cycle input $I \triangleq \{\langle p, v\rangle \mid p \in P$, v is the concrete value of $p\}$ of a design is the set of inputs with their concrete values for a clock cycle.

Definition 2.4 (Test Case)**.** A test case $\mathcal{T} \triangleq I_1, I_2, \ldots, I_n$ of a SystemC design is a sequence of cycle inputs, such that cycle input I_i $(1 \leq i \leq n)$ will be applied to the design at clock cycle i as inputs.

Chapter 3

SYMBOLIC EXECUTION OF SYSTEMC DESIGNS

Simulation-based approach is the "workhorse" for SystemC verification [96], which requires a set of concrete test cases. Traditionally, test cases are manually written or randomly generated. Manual test writing requires indepth knowledge of a DUV, which is time-consuming, labor-intensive, and error-prone. Random test generation, in contrast, is fast. However, random testing may generate many redundant test cases that result in long simulation time. Moreover, random testing usually leaves hard-to-reach segments and corner cases unexplored where bugs are likely to appear and hardware Trojans are often hidden. This chapter presents our novel approach to test generation of SystemC designs using symbolic execution, namely SESC [71].

3.1 OVERVIEW

Figure 3.1 shows the workflow of SESC. It has three key steps: (1) test-harness generation, (2) symbolic execution, and (3) test-case generation. For a given SystemC design, its test harness is generated and compiled with the design together to LLVM bitcode [61]. The symbolic execution engine takes the LLVM bitcode as input and executes it symbolically to explore as many execution paths as possible. When an execution path terminates or encounters an error, SESC sends the path constraints that are represented by symbolic expressions to a constraint solver,

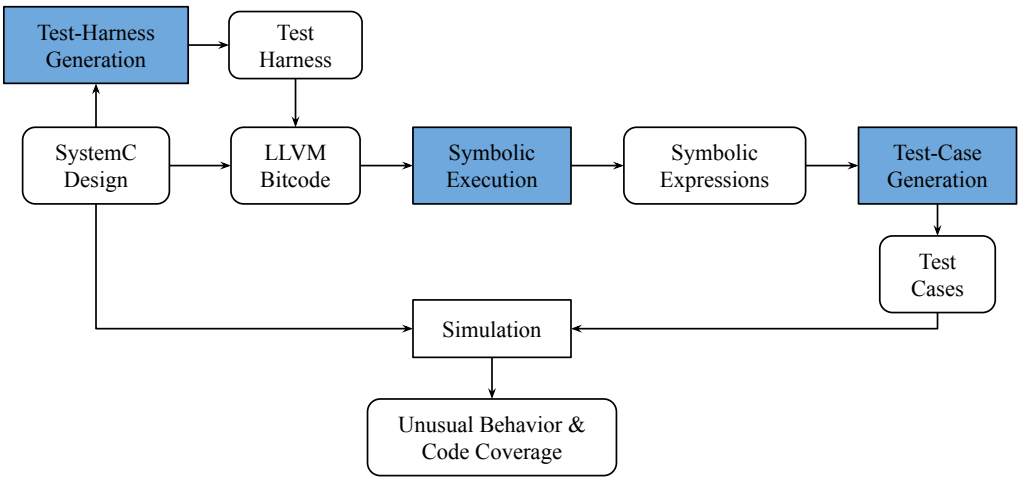

Figure 3.1: Workflow of SESC

which returns concrete values that satisfy the expressions if possible. Then SESC generates a concrete test case for the path. The generated test cases are then simulated on the SystemC design to detect unusual behaviors and compute coverage statistics.

SESC executes SystemC designs symbolically using KLEE [8] symbolic execution engine. KLEE targets sequential C programs and is not designed to execute SystemC designs where SystemC designs have specific characteristics. Therefore, we need to adapt KLEE to execute SystemC designs efficiently and provide more hardware specific semantics.

3.2 TEST-HARNESS GENERATION

SystemC designs usually include hierarchical structures, object-oriented features, and hardware-oriented data types. Generally, a SystemC design also contains multiple processes that run concurrently, which requires a scheduler to simulate the design. These features are implemented in the SystemC library. Thus, a

SystemC design by itself is not a stand-alone program. It invokes the SystemC library and communicates with its environment during simulation. Therefore, a test harness that models the environment must be provided to enable symbolic execution of SystemC designs. Although the SystemC library can serve as the test harness for all designs, it is impractical because the SystemC library is too complex for symbolic execution directly. A key challenge here is how to generate a test harness that should be simple enough so that SESC can efficiently execute a SystemC design symbolically.

Currently, we generate a test harness manually for a SystemC design. A test harness includes global variables definitions, synchronization mechanisms, symbolic variables constructions, and process registrations. Figure 3.2 shows the skeleton of the test harness for the design shown in Figure 2.2. Shared signals are defined as global structure *globalVars*. It only has one member *data* in this case. The harness defines two variables of type *globalVars*. The variable *currState* contains the synchronized value, while *LStates* is an array that each element is modified by one process. The function `SESC_make_symbolic(···)` constructs symbolic variables and `SESC_thread(···)` registers a SystemC process with SESC so that SESC can schedule it when required. The function `SESC_start`(*numCycles*) starts symbolic execution, where *numCycles* specifies how many clock cycles to simulate.

3.3 SCHEDULER

SystemC is widely used to model concurrent systems consisting of multiple processes. To deal with the SystemC concurrency, we have implemented a scheduler in

```
1  typedef struct Globals{
2    int data;
3  } globalVars;
4
5  globalVars currState, LStates[2];
6  ......
7  void PrePrOcess(currState) { ...... }
8  void Sync(LStates) { ...... }
9
10 int main(int argc, char **argv) {
11   ......
12   SESC_make_symbolic(&en, sizeof(en), "en");
13   SESC_make_symbolic(&din, sizeof(din), "din");
14
15   SESC_thread("P1", &en, &clk, &din, &LStates[1].data, &dout);
16   SESC_thread("P2", &en, &clk, &din, &LStates[2].data, &dout);
17   ......
18   SESC_start(numCycles);
19
20   return 0;
21 }
```

Figure 3.2: Skeleton of the test harness for the design shown in Figure 2.2

Algorithm 1: SYM-EXE-SCHEDULER(P, $numCycles$)

Data: $currState$ and $LStates$ are global variables.

Result: scheduling the symbolic execution of processes P for $numCycles$ cycles.

```
1  cycles ← 0
2  runnable ← ∅
3  foreach p ∈ P do
4      ENQUEUE(runnable, p)        ▷ puts each process into the runnable queue
5  while cycles ≤ numCycles do
6      next_runnable ← ∅
7      LStates ← PREPROCESS(currState)
8      while runnable ≠ ∅ do
9          q ← DEQUEUE(runnable)                ▷ q is a runnable process
10         SYM-EXE-PROCESS(q)                   ▷ process q is invoked to run
11         ENQUEUE(next_runable, q)
12     runnable ← next_runnable
13     currState ← SYNC(LStates)                ▷ synchronize shared variables
14     cycles ← cycles + 1
```

SESC to manage multiple processes, as shown in Algorithm 1. Our symbolic execution engine supports two types of processes, SC_THREAD and SC_CTHREAD, in the high-level synthesizable subset of SystemC.

According to the SystemC specification [49], access to shared storage should be

synchronized explicitly to avoid non-deterministic behavior, although the scheduler is non-deterministic. Thus, different scheduling sequences should not affect the simulation result for a well-formed design, which means that it is sufficient for the design to be simulated by only one scheduling sequence for each clock cycle. Therefore, we simulate designs deterministically using symbolic execution and synchronize shared storage explicitly. If a design is not well-formed, such as variables other than signals are used as inter-process communication, there are potential races. In this work, we assume that there are no races for a DUV.

As shown in Algorithm 1, before execution starts, the scheduler initializes simulation cycle as zero and puts all runnable processes into the runnable queue *runnable*. When execution starts, the scheduler first empties the queue *next_runnable* at the beginning of each clock cycle. Then, PrePROCESS is called to make N replicas of *currState* and store them in the array *LStates*, where N is the number of processes. After that, the scheduler removes each process from *runnable*, and calls Sym-Exe-Process to execute the selected process symbolically. Note that each process modifies its local state. When the process encounters `wait`, the scheduler puts the process into *next_runnable*. When *runnable* is empty, the execution is finished for this clock cycle. So the scheduler puts every process into *runnable* for the next clock cycle and synchronizes all local states resulting in a new global state *currState*, followed by advancing the simulation cycles. If the number of simulation cycles reaches *numCycles*, the simulation is finished, and the execution engine terminates.

3.4 SYMBOLIC EXECUTION OF SYSTEMC DESIGNS

To symbolically execute SystemC designs, besides the concurrency addressed in the previous section, SESC needs to address the following three technical challenges.

First, the path explosion problem is a major issue of symbolic execution to explore a complex program thoroughly. The number of paths is approximately exponential to the number of branches in a program. Not surprisingly, this problem also exists with symbolic execution of SystemC designs.

We apply two bounds to curb this problem. One is the time bound that is the maximum time for symbolic execution engine to run. The time bound ensures that SESC will terminate in a given amount of time. If SESC does not finish within the given time, there may be unfinished paths. For such paths, SESC still generates test cases with the path constraints collected before termination. The other bound is the clock cycle bound that specifies how many clock cycles to simulate.

Second, a SystemC design has a hierarchical modular structure usually and includes the object-oriented features, such as inheritance and polymorphism. Our approach flattens these features first by preprocessing SystemC designs. Then, they are compiled to LLVM bitcode using the *clang* compiler [21]. Each process is flattened as a function whose name is the same as the process name. All the input and output ports of the module, as well as shared signals among processes, become pointer parameters of the function.

Figure 3.3 illustrates the skeleton of the flattened result for the design shown in Figure 2.2. Processes P1 and P2 are interpreted as function P1 and P2. The input ports *en, clk*, and *din*, output port *dout*, and shared signal *data* become the pointer parameters of the functions.

Third, a SystemC design may contain hardware-oriented data structures such

```
1 void P1(*en, *clk, *din, *data, *dout) {
2   ......
3 }
4
5 void P2(*en, *clk, *din, *data, *dout) {
6   ......
7 }
```

Figure 3.3: Skeleton of flattened result for the design shown in Figure 2.2

as port, signal, FIFO, arbitrary-width data, and bit-precise operation. We address these structures by either implementing their stubs or adapting the symbolic execution engine. Ports are interpreted as function parameters as described above. Predefined channels such as *sc_signal* and *sc_fifo* are replaced by our implementations. Arbitrary-width data types and bit-precise operations, such as bit selection and bit set, are ubiquitous in hardware designs. KLEE does not support such data types and bit operations. We have implemented the functionalities in our symbolic execution engine, which supports arbitrary-width data types and bit-precise operations.

With the aforementioned challenges addressed, a SystemC design can be executed symbolically now. SESC collects path constraints along the execution of each path. For the example shown in Figure 2.2, suppose we set the clock cycle bound as three and the corresponding symbolic inputs for three clock cycles are $\{en_1,\ din_1\}$, $\{en_2,\ din_2\}$, and $\{en_3,\ din_3\}$. After symbolic execution of the design, SESC can collect constraints for all explored paths. Three sample constraints are

as follows.

$Constraint\ 1 : en_1 = 0\ \wedge en_2 \neq 0\ \wedge din_2 < 0 \wedge en_3 \neq 0\ \wedge din_3 < 0$

$Constraint\ 2 : en_1 = 0\ \wedge en_2 \neq 0\ \wedge din_2 > 0 \wedge (din_2 \% 2 \neq 0) \wedge en_3 \neq 0\ \wedge din_3 < 0$

$Constraint\ 3 : en_1 = 0\ \wedge en_2 \neq 0\ \wedge din_2 > 0 \wedge (din_2 \% 2 = 0) \wedge en_3 = 0$

3.5 TEST-CASE GENERATION

The path constraints collected by SESC are denoted as symbolic expressions. However, symbolic expressions are hard to understand for general designers. So when an execution path terminates, the symbolic expressions are sent to a constraint solver, which returns concrete values that satisfy the expressions if possible. As the constraints shown at the end of previous section, the corresponding sample test cases generated by SESC are as follows.

$\mathcal{T}_1 \triangleq I_1, I_2, I_3$ where $I_1 \triangleq \{\langle en_1, 0\rangle,\ \langle din_1, 0\rangle\},\ I_2 \triangleq \{\langle en_2, 1\rangle,\ \langle din_2, -1\rangle\},$ $I_3 \triangleq \{\langle en_3, 1\rangle,\ \langle din_3, -1\rangle\}$.

$\mathcal{T}_2 \triangleq I_1, I_2, I_3$ where $I_1 \triangleq \{\langle en_1, 0\rangle,\ \langle din_1, 0\rangle\},\ I_2 \triangleq \{\langle en_2, 1\rangle,\ \langle din_2, 1\rangle\},$ $I_3 \triangleq \{\langle en_3, 1\rangle,\ \langle din_3, -1\rangle\}$.

$\mathcal{T}_3 \triangleq I_1, I_2, I_3$ where $I_1 \triangleq \{\langle en_1, 0\rangle,\ \langle din_1, 0\rangle\},\ I_2 \triangleq \{\langle en_2, 1\rangle,\ \langle din_2, 2\rangle\},$ $I_3 \triangleq \{\langle en_3, 0\rangle,\ \langle din_3, 0\rangle\}$.

3.6 EXPERIMENTAL RESULTS

This section describes our experiments on 11 high-level synthesizable SystemC designs, as listed in Table 3.1. The first column gives the names of the designs. The second column shows the number of processes of each design. Lines of code (LoC) for each design are listed in the third column. LoC is calculated using `cloc` [24]. All experiments were performed on a desktop with 4-core Intel(R) Xeon(R) CPU, 8 GB of RAM, and running the Debian Linux operating system with 64-bit kernel version 3.16.

Table 3.1: Summary of 11 SystemC designs

Designs	**# of Process**	**LoC**
RISC_CPU_exec	1	126
RISC_CPU_control	1	826
RISC_CPU_bdp	3	148
UsbArbStateUpdate	2	85
ADPCM	1	134
IDCT	1	244
Sync_mux81	1	52
MIPS	1	255
RISC_CPU_mmxu	1	187
RISC_CPU_crf	5	927
RISC CPU	13	2056

Note that the design `RISC CPU` consists of 10 modules, 13 processes, and 2056 LoC in total. Figure 3.4 shows the architecture of the CPU design. The CPU reads program instructions and executes them and then writes the results back to registers or data memory. The instruction set is defined based on commercial RISC

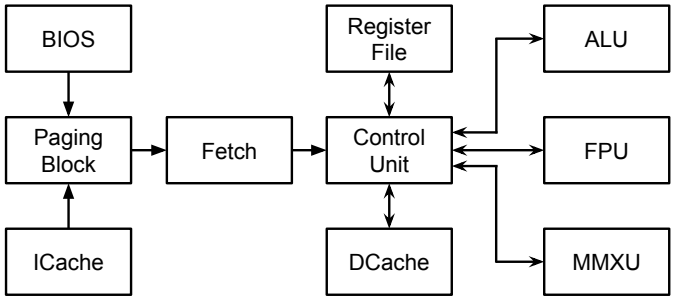

Figure 3.4: Architecture of RISC CPU

processor together with MMX-like instructions. `BIOS` stores system bios data. `ICache` caches program instructions, while `DCache` caches data. `Fetch` fetches instructions from `Paging Block`. `Control Unit` decodes instructions. `Register File` models registers. `ALU` is an integer execution unit, while `FPU` is a floating point execution unit. `MMXU` is an MMX-like execution unit.

3.6.1 Coverage Methodology

Appropriate coverage metrics are required to evaluate the effectiveness and confidence in the verification results. Functional coverage needs to be redeveloped for each design and built by engineers with indepth knowledge of both the design specification and implementation. Thus, it is not automatic. SystemC is widely used for modeling functionalities of systems at ESL. Therefore, we adopt the typical code coverage, line coverage and branch coverage, which are widely used and understood. We choose line and branch coverage reported by `LCOV` [62]. SystemC designs heavily use the predefined structures and facilities provided by the SystemC library. `LCOV` reflects the library code in the design code by default. For example, `LCOV` reports two branches for the library call `wait()`, while there is no branch from a design point of view. To this end, we utilize three exclusion markers provided

Table 3.2: Time and memory usage, and coverage results

Designs	**Time(s)**	**MEM(MB)**	**# of TCs**	**LCov(%)**	**BCov(%)**
RISC_CPU_exec	3.23	46.9	35	100	100
RISC_CPU_control	0.57	17.8	76	100	100
RISC_CPU_bdp	0.15	17.5	36	100	100
UsbArbStateUpdate	0.05	13.7	10	100	100
ADPCM	1.88	16.2	25	100	100
IDCT	180	134.0	135	100	100
Sync_mux81	0.04	13.5	10	100	100
MIPS	178.23	27.6	39	100	97.9
RISC_CPU_mmxu	11.38	15.6	95	99.4	97.9
RISC_CPU_crf	300	61.1	1759	98.2	95.7
RISC CPU	169	264	2099	96.3	93.2

by `LCOV` to exclude specific lines from branch coverage, since we only calculate the actual branches in a design itself. The marker `LCOV_EXCL_BR_LINE` excludes a single line from branch coverage, while the pair of markers, `LCOV_EXCL_BR_START` and `LCOV_EXCL_BR_STOP`, exclude multiple consecutive lines from branch coverage. We also exclude test harness code when calculating code coverage. The rules are applied to all code coverage computed in this dissertation research.

The experimental results are demonstrated in Table 3.2. Column 2 and column 3 show the time and memory usage, which is modest. The fourth column gives the number of generated test cases. The code coverage achieved by SESC is presented in the last two columns. As we can see from the table, our approach can achieve 100% line and branch coverage for most designs. There are two possible reasons that SESC does not achieve 100% code coverage for some designs. First, compiler

may perform optimization when compiling from SystemC code to LLVM bitcode. Second, there may be some unreachable statements and branches so that 100% code coverage can never be achieved.

3.6.2 Comparison with Random Testing

We would like to compare the results of SESC with the state-of-the-art approaches. However, to the best of our knowledge, no other existing SystemC verification approaches provides such code coverage. Therefore, we compare the code coverage results of SESC with two groups of random testing. They are denoted as Random10 and Random100 that represent the number of random tests with 10 times and 100 times of the number of test cases generated by SESC. For each group, we conducted experiments 10 times and calculated the average. The comparisons of line coverage and branch coverage are shown in Figure 3.5 and Figure 3.6, respectively. As we can see from the figures, SESC gains the best coverage results. Especially, our approach beats both by a significant margin for designs `MIPS`, `IDCT` and `Sync_mux81`.

Based on the comparisons, the SystemC designs can be divided into three groups. In the first group, designs can be easily achieved high code coverage using random testing, such as `ADPCM`. Designs in this group contain only a few branches. The second group contains designs that can be achieved relatively high code coverage by random testing with much more test cases, such as `RISC_CPU_control`. Designs in this group consist of relatively more branches than the first group. However, there are barely any nested branches. In the third group, it is hard to achieve high code coverage using random testing even with much more test cases, such as `MIPS`. This is mainly because designs in this group include nested branches and compound conditions of branches. As we can see, SESC is able to achieve very

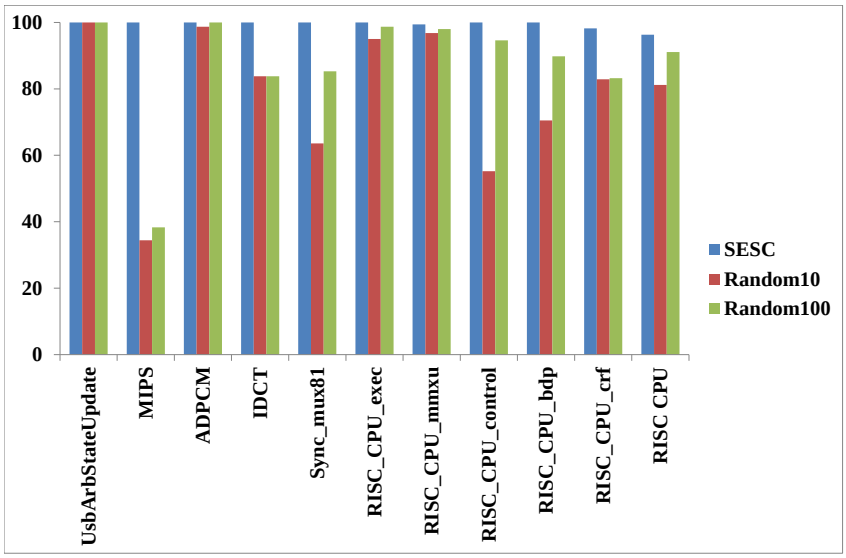

Figure 3.5: Line coverage of SESC testing vs. Random10 and Random100 for 11 SystemC designs. SESC beats both by as much as 66%.

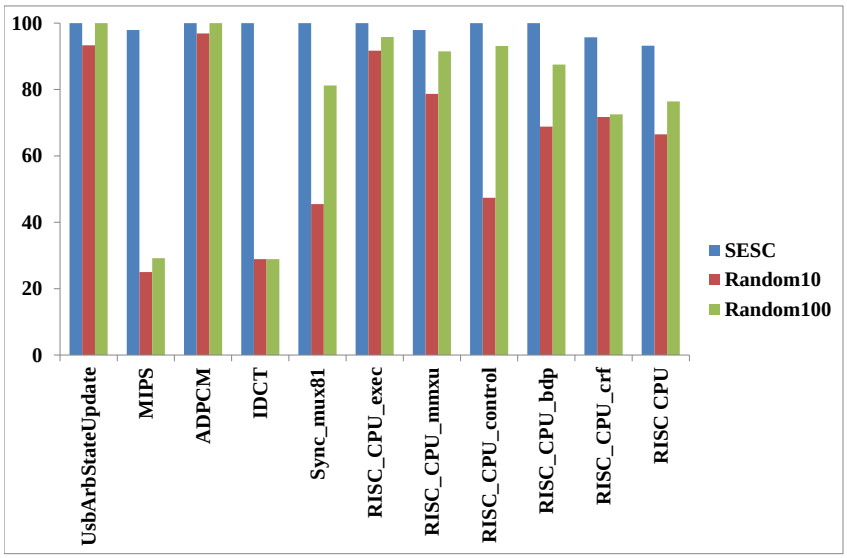

Figure 3.6: Branch coverage of SESC testing vs. Random10 and Random100 for 11 SystemC designs. SESC beats both by as much as 71%.

high code coverage for relatively complex designs, while random testing cannot.

3.7 SUMMARY

It is very challenging to generate high coverage test cases automatically for SystemC designs due to the complexity of SystemC designs. In this chapter, we have presented an approach to generating high coverage test cases automatically using symbolic execution. We have implemented the proposed approach as a prototype, namely SESC. We have also applied SESC to a set of SystemC designs to evaluate its performance. The results of our experiments demonstrate that SESC is able to generate test cases that achieve high code coverage with modest time and memory usage, as well as scaling to designs with practical sizes.

Chapter 4

CONCOLIC TESTING OF SYSTEMC DESIGNS

The previous chapter presents an automated test generation approach for SystemC designs using symbolic execution. It has two main limitations. First, it is focused on high-level synthesizable SystemC designs, which is a subset of SystemC designs. Second, it is time-consuming to model hardware-oriented data structures in the SystemC library case by case. In this chapter, we describe our improved approach, namely concolic testing of SystemC designs (CTSC) [70], which has better scalability and requires much less human effort. Moreover, we adopt the ABV techniques to determine whether a test case passes or fails according to whether or not it triggers an assertion.

4.1 OVERVIEW

Figure 4.1 shows the workflow of our concolic testing of SystemC designs in the binary level. It has four key steps: (1) testbench generation, (2) concolic test generation, (3) test-case selection, and (4) testing with generated test cases. For a given SystemC design, its testbench is generated first and compiled with the design into a binary by linking the SystemC library. Then, the binary and an initial test case, called the *seed*, are fed to a binary concolic execution engine that includes two parts, concrete execution and symbolic execution. Concrete execution simulates a design concretely during which an execution trace is obtained. Symbolic execution

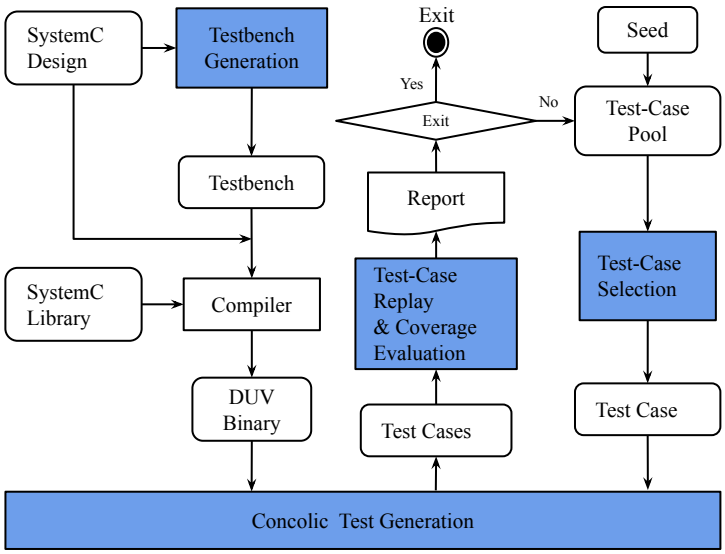

Figure 4.1: Workflow of concolic testing of SystemC designs

explores the trace symbolically to generate new test cases. Subsequently, the newly generated test cases are simulated on the SystemC design to detect design errors, as well as generating coverage information. Afterwards, a test case is selected from the newly generated test cases for a new iteration of concolic test generation. This process repeats until a termination condition is achieved.

Algorithm 2 illustrates the validation process using concolic testing. For most complex system designs, due to path explosion, the common usage model for automated test generation by symbolic or concolic execution is to run for a fixed amount of time or run until a target coverage goal is reached. We also follow such a model. SC-Con-Testing takes four parameters, a SystemC design *duv*, an initial test case *seed*, a target coverage *tgt*, and a time bound β, as inputs. The outputs are the generated test cases and the validation results that includes achieved coverage and bug information if any.

Here, TC saves all generated test cases and TCP is a test-case pool accessed dynamically during the validation process. At the beginning, the concolic execution

Algorithm 2: SC-Con-Testing(duv, $seed$, tgt, β)

```
1  TC ← {seed},  TCP ← {seed}
2  cov ← 0,  report
3  while (TCP ≠ ∅) ∧ (cov < tgt) ∧ (time < β) do
4      τ ← Test-Case-Selection(TCP)
5      TCP ← TCP \ {τ}
6      NTC ← Concolic-Test-Gen(duv, τ)
7      foreach t ∈ NTC do
8          report ← Test-Case-Replay(duv, t)
9          TCP ← TCP ∪ {t}
10     TC ← TC ∪ NTC
11     cov ← Coverage-Evaluation()
12 return TC, cov,  report
```

engine executes the design with the *seed* (line 6). When the engine terminates, it generates new test cases that are saved in a temporary set, NTC. For each newly generated test case, our framework reruns it on the SystemC design (line 8) to look for unusual behavior, such as assertion violations, and generate information for computing coverage. Each test case that is replayed is added to TCP (line 9) for the next iteration. Subsequently, the newly generated test cases are added to TC (line 10), and Coverage-Evaluation is called to compute coverage (line 11). If the coverage satisfies the coverage target tgt, the concolic testing terminates. Otherwise, the test-case selector selects a new test case (line 4) and removes it from TCP (line 5). Then, the binary concolic execution engine runs again. This process repeats until the target or other termination conditions are achieved. The time

bound β guarantees the termination of the validation process in case the target coverage cannot be achieved within the given time. The variable *time* denotes the total time elapsed since the start of the validation process. In the following, we will discuss the details of each key step.

4.2 TESTBENCH GENERATION

A testbench in this approach is different from a test harness in Chapter 3. A testbench is to generate stimuli and apply them to a design, as well as recording and monitoring the output of a design. In addition to these, a test harness in SESC also includes synchronization mechanisms, process registrations and other environment modeling. Thus, a testbench is simpler than a test harness. To apply CTSC to an existing SystemC project, the existing testbench of the SystemC project can be reused with slight modification. Instead of generating concrete stimuli, the function `CTSC_make_concolic` is used to construct stimuli as symbolic in addition to keeping their concrete values, so-called *concolic stimuli*. Additionally, we have developed a GUI to generate a testbench for a DUV automatically. Users simply specify names and types for both stimuli and outputs of DUVs. Users can also set the properties of a clock signal, such as period and duty cycle. A complex testbench may demand slight modification manually.

Figure 4.2 illustrates the stimuli generation module in a testbench for the design shown in Figure 2.2. The module has one clock input *clk*, and two outputs, *en* and *data* that are connected to the inputs of the design. The function `CTSC_make_concolic` constructs concolic stimuli for the design. The recording and monitoring of the output and the definition of the clock signal are straightforward. Thus, they are not presented.

```
1  SC_MODULE(driver) {
2    sc_in<bool> clk;
3    sc_out<bool> en;
4    sc_out<int> data;
5
6    void run() {
7      int data_tmp;
8      bool en_tmp;
9
10     wait();
11     while (true) {
12       CTSC_make_concolic(&data_tmp, sizeof(data_tmp), "data_tmp");
13       CTSC_make_concolic(&en_tmp, sizeof(en_tmp), "en_tmp");
14
15       data.write(data_tmp);
16       en.write(en_tmp);
17
18       wait();
19     }
20   }
21
22   SC_CTOR(driver) {
23     SC_CTHREAD(run, clk.pos());
24   }
25 };
```

Figure 4.2: An example of stimuli generation module for the design shown in Figure 2.2

4.3 CONCOLIC TEST GENERATION

Algorithm 3 illustrates the steps to generate test cases. First, a SystemC DUV is simulated with a concrete test case by invoking CONCRETE-EXECUTION(duv, τ). Concrete execution follows the scheduling mechanism provided by the SystemC library. During concrete execution, the current execution path is recorded as a trace tr. This way, a concurrent SystemC design is unwound as a sequential self-contained execution trace. Since the inputs of a DUV are made symbolic in addition to concrete values, the symbolic property of an input is also captured in the trace, which enables test generation afterwards. The recorded trace includes all information such as executed instructions that is required by symbolic execution subsequently.

After the trace is generated, it is explored symbolically to generate new test cases. Line 2 initializes the execution state s with the generated trace tr. An execution state includes a stack, a heap, concrete values of the inputs, path conditions (represented as symbolic expressions), a register file, and a program counter. TCS stores the generated test cases and is initialized to empty (line 3). Line 4 – 16 demonstrate the symbolic execution for test generation. If there is an instruction for execution (line 4), the instruction I is fetched (line 5) and executed (line 6). If it is a branch instruction (line 7), the symbolic predicate bp of I is computed (line 8). Then, the results of the symbolic predicate is computed with concrete values from τ (line 9). If the value of tb is `true` for the branch, the true path of this branch is taken in the concrete execution. Therefore, a test case is generated for the false path (line 11). Otherwise, a test case is generated for the true path (line 13). If the newly generated test case tc is not redundant, it is saved in TCS (line 14 – 15). Subsequently, line 16 sets the next instruction to be executed. If

Algorithm 3: CONCOLIC-TEST-GEN(duv, τ)

```
1  tr ← CONCRETE-EXECUTION(duv, τ)
2  s ← STATE-INIT(tr)
3  TCS ← ∅
4  while HAS-NEXT-INSTRUCTION(s) do
5      I ← GET-NEXT-INSTRUCTION(s)
6      EXECUTE-INSTRUCTION(I)
7      if I is branch then
8          bp ← GET-SYMBOLIC-BRANCH-PREDICATE(I)
9          tb ← COMPUTE-TAKEN-BRANCH(bp, τ)
10         if tb == true then
11             tc ← GENERATE-TEST-CASE(bp, false)
12         else
13             tc ← GENERATE-TEST-CASE(bp, true)
14         if tc ∉ TCS then
15             TCS ← TCS ∪ tc
16     SET-NEXT-INSTRUCTION(s)
17 return TCS;
```

every instruction of s has been executed, current iteration is complete. Finally, all generated test cases are returned (line 17).

Note that a SystemC program is a hardware design that can be simulated for an arbitrary number of clock cycles. Therefore, a concrete execution trace of a SystemC DUV can be intimidatingly long. In our approach, we simulate a SystemC DUV for a fixed number of clock cycles.

4.4 TEST-CASE SELECTION

Concolic test generation requires a concrete test case for each iteration. At the beginning, there is only one test case, the *seed*, which is simply selected by Test-Case-Selection. After the first iteration, multiple test cases may be generated. Thus, different test-case selection strategies can be adopted. Currently, we have developed three test-case selection strategies in terms of the time stamp of test cases: (1) first-come-first-serve (FCFS) that the earliest generated test case is selected first; (2) last-come-first-serve (LCFS) that the last generated test case is selected first, and (3) random selection that a test case is selected among all generated test cases randomly. In the future, we will use SystemC features to guide test-case selection. In addition, machine learning may play an important role in test-case selection.

4.5 TESTING WITH GENERATED TEST CASES

It is only half of the story to generate test cases that explore as many paths as possible. A generated test case follows the exact same code path that the concolic execution engine explored. Thus, if an error is encountered by the concolic execution engine, the generated test case can reproduce the error afterwards. Therefore,

our framework validates SystemC designs by replaying the generated test cases (line 8 in Algorithm 2). This replay is checked for unusual behavior or errors. If TEST-CASE-REPLAY detects an error when replaying a test case, it saves the test case and generates a report that records the detailed information of the error, such as error type and error location. This helps users better analyze the design and fix the error.

To check whether a test case passes or fails, we have also integrated the ABV techniques in which designers use assertions to capture specific design intents. Assertions are used to improve the ability to observe bad behavior once they are triggered by specific test cases. By getting an assertion triggered, users can easily identify if there are bugs or invalid inputs. This helps users fix the bug or further constrain the input ranges.

Currently, our framework is focused on generating test cases, not focused on generating assertions for designs. Instead, we utilize the existing assertions in the SystemC designs to evaluate the effectiveness of the CTSC-generated test cases.

4.6 EXPERIMENTAL RESULTS

We have implemented our approach in a prototype where our binary concolic execution engine is based on CRETE [10], a recently open-sourced concolic execution engine that targets software programs. This section presents our experimental results of CTSC on 19 total SystemC designs. The summary of 19 designs is shown in the first three columns of Table 4.1. It lists the names of the designs, the number of processes, and LoC, respectively. All experiments were performed on a desktop with a 4-core Intel(R) Core(TM) i7-4790 CPU, 16 GB of RAM, and running the Ubuntu Linux OS with 64-bit kernel version 3.19.

Table 4.1: Summary of designs, time and memory usage

Designs	**# of Proc.**	**LoC**	**ET (s)**	**MEM (MB)**	**# of TCs**
DES	14	2401	186	3928	35
RISC_CPU_bdp	3	148	3077	505	149
RISC_CPU_mmxu	1	187	1291	240	258
RISC_CPU_exec	1	126	251	230	44
RISC_CPU_floating	1	127	576	243	122
Qsort	1	86	88	175	41
UsbTxArbiter	5	144	234	265	298
Sync_mux81	1	52	73	180	28
MIPS	1	255	207	124	474
IDCT	1	244	335	574	494
MD5C	1	271	21	465	37
RSA	1	324	1944	8103	131
RISC_CPU_crf	5	927	10863	331	1220
RISC_CPU_control	1	826	12005	347	1246
ADPCM	1	134	25	220	40
Y86	11	301	493	480	67
Pkt_switch	17	376	3189	333	385
RISC CPU	13	2056	17520	1303	386
Master/Slave Bus	5	974	24	205	88

4.6.1 Code Coverage Improvement

In our experiments, we developed the seed for each design. We set a 24-hour time bound and 100% branch coverage target for all designs. The actual execution time (ET) of each design in seconds is listed in the fourth column of Table 4.1, after which the branch coverage could not be improved within the 24-hour time bound. The corresponding maximum memory usage and the number of generated test cases are presented in Columns five and six. As shown, the time and the memory usage was modest. Since `RSA` and `DES` are cipher algorithms that do computation on large numbers, they used more memory.

Figure 4.3 and Figure 4.4 show the code coverage improvement on 19 total designs over the seeds with the time usage listed in Table 4.1. In our current experiments, we adopted FCFS test-case selection strategy. In the future, we will evaluate the effects of different test-case selection strategies. As illustrated, the CTSC-generated test cases are able to improve code coverage substantially. For most designs, high code coverage is achieved in a short time. CTSC achieves 100% line coverage on ten designs and 100% branch coverage on eight designs. Table 4.2 shows the overall code coverage improvement of CTSC over the seeds. On average, CTSC achieves 97.3% line coverage and 91.8% branch coverage (Column 2 of Table 4.2). The maximum improvement of line and branch coverage are 84% and 91.5% (Column 3 of Table 4.2), respectively. On average, line and branch coverage are improved by 32.3% and 50.2% (Column 4 of Table 4.2), respectively.

Besides the final coverage our approach achieved, we also graphically demonstrate the cumulative progression as test cases were generated. Here, we selected three designs, `RISC CPU`, `RISC_CPU_control`, and `RISC_CPU_crf`, which use relatively longer time. Figure 4.5 and Figure 4.6 illustrate the cumulative progression

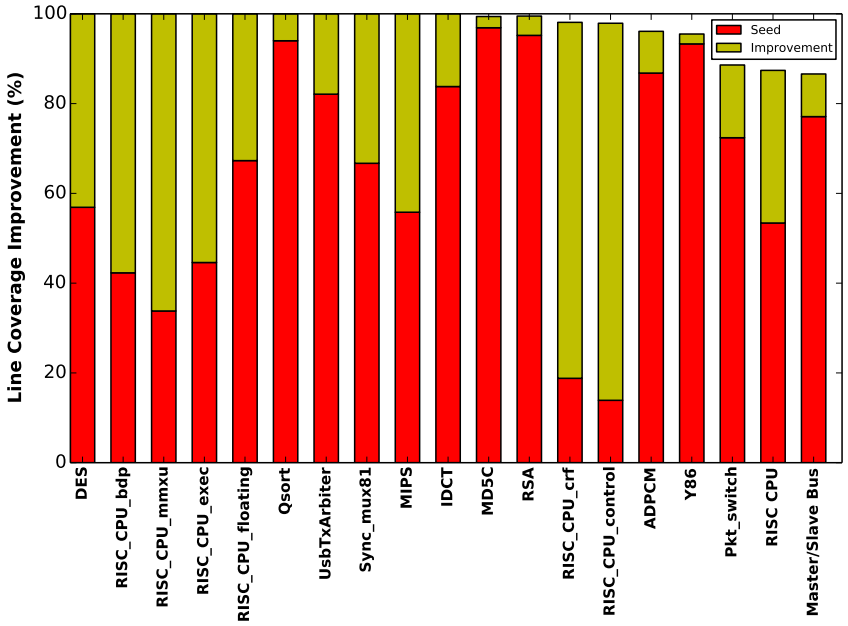

Figure 4.3: Line coverage improvement on 19 total designs

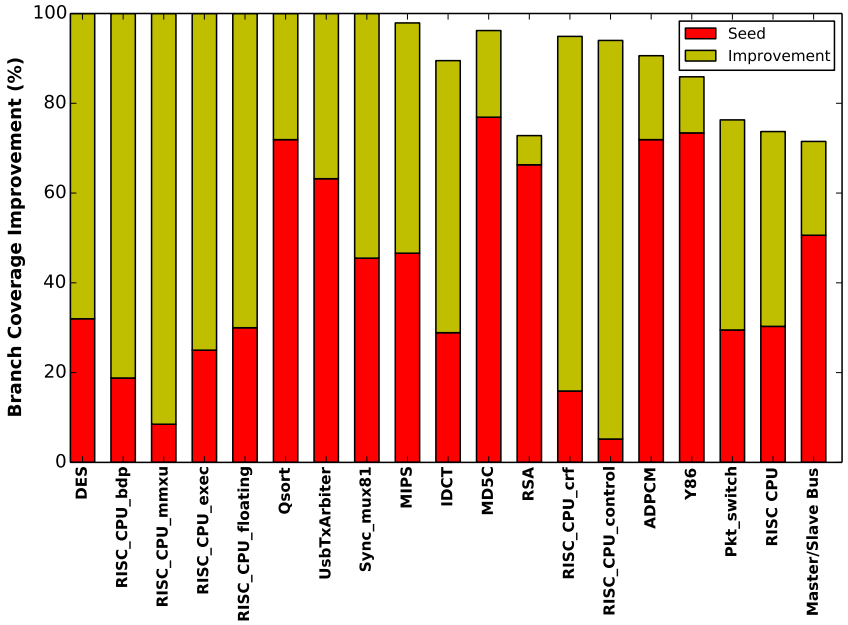

Figure 4.4: Branch coverage improvement on 19 total designs

Table 4.2: Coverage improvement over seeds

Coverage	**Ave. (%)**	**Max Δ (%)**	**Ave. Δ (%)**
Line	97.3	84	32.3
Branch	91.8	91.5	50.2

of line coverage and branch coverage, respectively. The figures demonstrate the 10-hour time line visually. As shown, the line coverage and the branch coverage are improved substantially within the first hour based on the seed, after which improvement tapers off in a few hours.

There are two possible reasons that our approach is unable to achieve 100% code coverage for some designs. First, the constraint solver may fail to solve complex symbolic expressions. Since there are symbolic inputs at each simulation cycle, the collected path constraints can be very complicated. Second, there are unreachable statements and branches in certain designs.

4.6.2 Comparison with Random Testing

We also compare the code coverage results of CTSC with random testing that randomly generates test cases automatically at each simulation cycle. We set the branch coverage of each design achieved by CTSC as the target for random testing. In case random testing could not achieve the target, we set a 24-hour time bound. We conducted the experiments of random testing 10 times for each design and computed the average. Note that the inputs of `Master/Slave Bus` has rigorous restrictions. It is hard to generate valid inputs randomly. Thus, we excluded this design.

The target is achieved on eight designs, but with many more test cases. Random

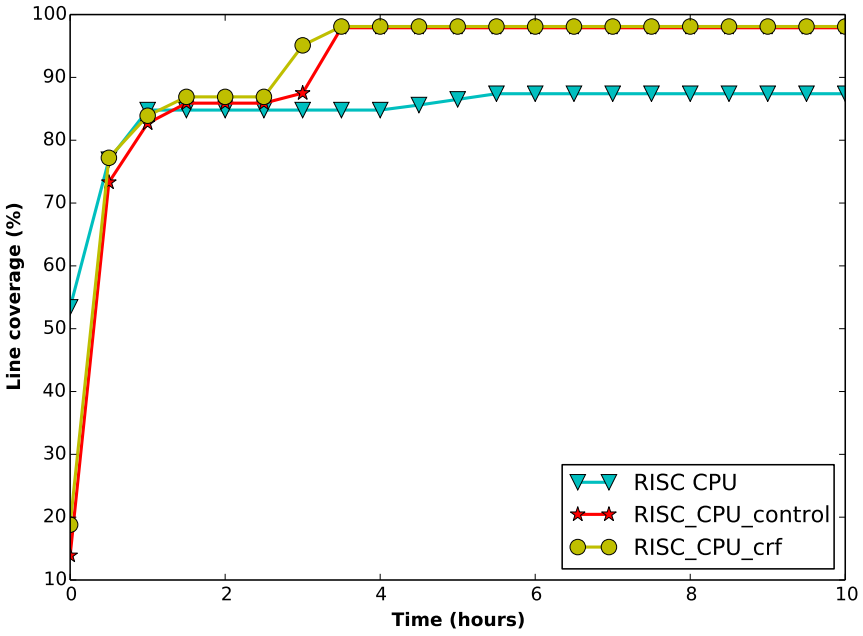

Figure 4.5: The cumulative progression of line coverage

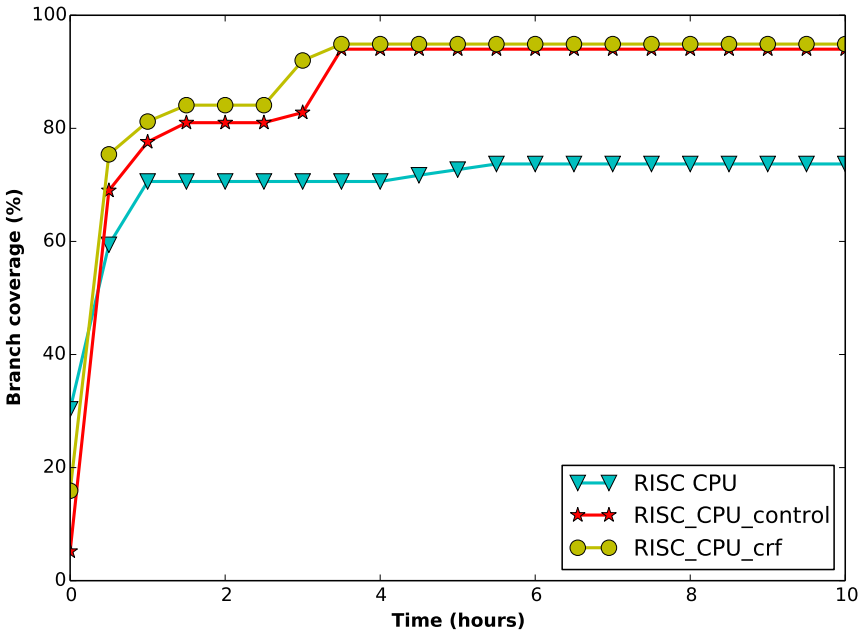

Figure 4.6: The cumulative progression of branch coverage

testing does not achieve the target for other ten designs after running 24 hours, although numerous random test cases were generated and simulated. Note that if regression testing is performed, it is time-consuming to simulate hundreds of thousands of test cases or more each time. Compared with random testing, CTSC has the advantage of generating much fewer test cases to achieve high code coverage and to cover corner cases efficiently where bugs are likely to appear, as illustrated in the following section.

4.6.3 Bug Detection

In addition to computing code coverage, we also show the capability of our approach to trigger assertions and detect bugs. When the validation process terminates, a validation report is generated automatically. The report is in a plain text format that mainly indicates the statuses of test cases (failure or pass) and assertion violations. Upon triggering an assertion, CTSC generates a test case automatically leading to the assertion, which helps designers find the root cause easier when debugging the design.

Among the 19 total designs, five designs contain assertions, as shown in the first column of Table 4.3. The second column shows the total number of assertions. The last three columns present the number of assertions triggered by the seed, by the CTSC-generated test cases, and by random testing, respectively. Although random testing can trigger some assertions, it generates many more test cases than CTSC. Note that some assertions always hold. For instance, although there are 15 total assertions in the design `RSA`, we have verified manually that 11 assertions can never be triggered. For example, the violation of assertion `assert(a == d)` directly following the assignment `a = d` cannot be triggered.

Table 4.3: Assertion coverage

Designs	**# of Assertions**			
	total	**by seed**	**by CTSC**	**by random**
RISC_CPU_exec	2	0	2	0
MD5C	1	0	1	0
RSA	15	0	3	3
Master/Slave Bus	11	0	6	N/A
RISC CPU	2	0	2	2

During our experiments, we found an interesting bug in the design `RSA`, an asymmetric cryptographic algorithm. The first step of `RSA` is to find two *different* large prime numbers, p and q. Note that the algorithm relies on the fact that p and q are different. If they are equal, the algorithm does not work correctly. However, this implementation does not check whether or not p and q are equal. In addition, we also found an out-of-bound access to an array in the design `Y86`. The bugs found by CTSC underlines the importance of performing automated concolic testing and the effectiveness of CTSC.

4.7 SUMMARY

As discussed in Chapter 3, SystemC designs are not stand-alone programs. SystemC simulation invokes libraries and communicates with its environment. To analyze the behavior of a SystemC design accurately, it often requires taking the dependent libraries into account. Due to the complexity of the SystemC library, most existing SystemC verification approaches either translate SystemC designs into other IRs, which can represent only a subset of SystemC usually, or handle the SystemC library by writing a simplified one. Thus, those approaches cover

only a subset of SystemC features. Although SystemC comes with a well-written user's manual and a reference implementation, it lacks formal specification and leaves out some implementation choices deliberately. Hence, even carefully writing a simplified library can easily result in a dialect. Moreover, such modeling is time-consuming, error-prone, and labor-intensive.

In contrast, our approach described in this chapter requires no translation of SystemC designs, no modeling of dependent libraries, and therefore supports all kinds of SystemC designs without restrictions. We have presented an automated, easy-to-deploy, and scalable concolic testing approach for SystemC designs, namely CTSC. The experimental results illustrate that CTSC is able to achieve high code coverage and detect bugs effectively, as well as scaling to designs with practical sizes. There are four major advantages of this approach. First, CTSC requires no translation of SystemC designs and no modeling of dependent libraries, while most existing work does. Second, CTSC supports *all* features of the SystemC language, while most existing approaches support only a subset of SystemC features. Third, CTSC provides an easy deployment model. It requires minimum engineering effort to apply CTSC to existing SystemC projects. Fourth, once a testbench is configured, the whole validation process is *fully automated*.

Chapter 5

HARDWARE TROJAN DETECTION IN SYSTEMC DESIGNS

5.1 MOTIVATION

Due to the growing complexities of SoCs and increasingly shortened time-to-market requirements, abstraction level of modern SoCs design has been raised from RTL to ESL, e.g., in C/C++ or SystemC. Modern SoCs design in ESL often include a large variety of behavioral IPs, such as microcontrollers, network processors, and digital signal processors. Developing and verifying all these IPs in-house is intimidating, if not impossible, due to time-to-market and budget constraints. New design paradigms such as outsourced design services and widely adoption of EDA tools have emerged. Although this new design trend tremendously improves modern SoCs design productivity, it results in partial relinquishment of the control over SoCs design process, which raises new hardware security issues such as hardware Trojan attacks in early design steps [83]. This becomes an emerging threat of computer security.

SystemC is a widely adopted ESL modeling language in the semiconductor industry. ESL SystemC designs usually serve as executable specifications for the subsequent SoCs design flow. Thus, it is critical to assure the trustworthiness of those ESL SystemC designs. If hardware vulnerabilities such as hardware Trojans are not discovered in ESL SystemC designs, they may be translated together with normal functionalities down to RTL and lower level implementations, which makes

them much more costly to fix. Existing hardware Trojan detection approaches, most of which are focused on RTL and lower levels, may be able to detect those hardware Trojans. However, detecting and fixing Trojans in RTL or lower levels is much more expensive than fixing them in ESL. Unfortunately, those low-level hardware Trojan detection approaches are not applicable to ESL SystemC designs since they have different characteristics from RTL and lower level implementations.

Until recently, there has only been a limited amount of research on hardware Trojan detection for ESL SystemC designs. The pioneering work [97] discusses hardware Trojan problem in behavioral designs and proposes a method to detect those Trojans using property checking. The subsequent work [63] uses coverage-guided fuzz testing to detect hardware Trojans in behavioral SystemC designs. Both approaches are focused on behavioral synthesizable SystemC designs, which are a subset of ESL SystemC designs. Our approach presented in the following does not have such a restriction so that it is applicable to any ESL SystemC design.

5.2 OVERVIEW

5.2.1 Threat Model

With the globalization of the semiconductor industry, modern SoC designers have been driven to outsource their IPs to reduce cost. In addition, the growing complexities of modern SoCs has raised the design abstraction level from RTL to ESL. As a result, modern SoCs design methodologies at ESL often involve integration of behavioral IPs supplied by third-party vendors, as well as intensive usage of EDA tools, to improve the design productivity. However, as shown in Figure 5.1, the trustworthiness of SoCs in ESL may be comprised. First, third-party IPs may

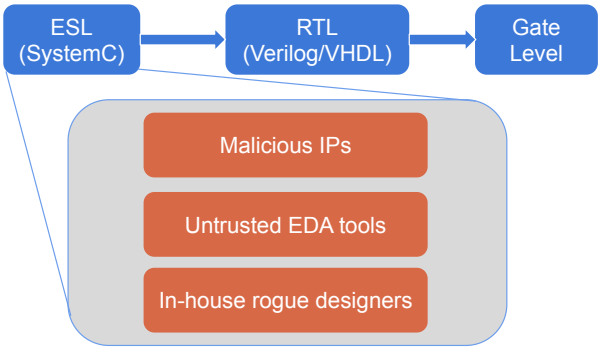

Figure 5.1: Adversarial threat model targeted by SCT-HTD

contain malicious implants. Although a testbench with test cases is likely provided with the IPs by third-party vendors, these test cases are not able to trigger the embedded Trojans. Second, untrusted EDA tools may also insert hardware Trojans into these behavioral designs. The research [81] has demonstrated that HLS tools can be leveraged to inject Trojans into resultant RTL implementations. Last but not least, in-house designers may leave back-doors when integrating SoCs, which makes the situation worse. Our approach mainly intends to detect hardware Trojans injected into ESL SystemC designs [66].

5.2.2 Workflow

In this work, we assume that there is a golden model for a behavioral SystemC DUV. A golden model is an executable behavioral model that is functionally correct and Trojan-free. Although it is very expensive, if not impossible, to develop an entire SoC in-house by designers, we argue that it should not be very time-consuming to develop a functionally correct golden model, which is not necessarily cycle accurate. As shown in Figure 5.2, our approach consists of two key components, selective concolic executor (SCE) and hardware Trojan detector (HTD). SCE generates test cases by selective concolic execution with coverage-guided state

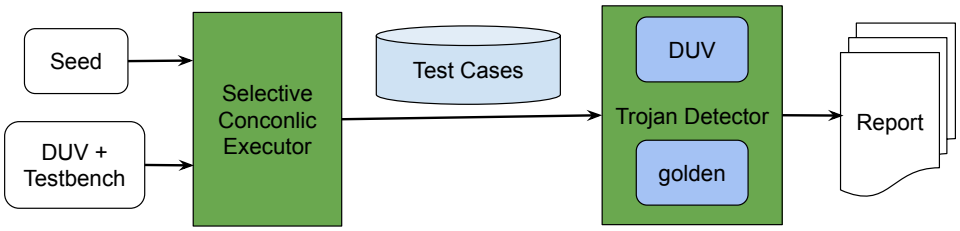

Figure 5.2: Selective concolic testing for hardware Trojan detection

search strategy, while HTD detects hardware Trojans by simulating the generated test cases on both the DUV and its golden model.

Algorithm 4 illustrates high-level steps of our proposed hardware Trojan detection approach. The algorithm HARDWARE-TROJAN-DETECTOR takes five parameters as inputs: a design under validation *duv*, an initial test case π (called *seed*), a testbench *tb*, a golden model *golden*, and a configuration file *config*. The set TC, which contains the seed initially (line 1), saves all generated test cases. The state s_0 is the initial execution state of the DUV with the seed, and assigned to s_n which is the next state to be explored by concolic execution (line 2). An execution state includes a stack, a heap, concrete values of the inputs, path conditions (represented as symbolic expressions), a register file, and a program counter. The queues FPS and SPS save execution states during the process (line 3). FPS, which includes the initial state at the beginning, saves first priority states that explore new code of interest, while SPS saves second priority states that do not explore new code of interest. The variable $INTS$ stores code ranges that users are interested in for hardware Trojan detection (line 4). Those code ranges are specified in the configuration file. A time bound β is also given in the configuration file (line 5), which guarantees the termination of the Trojan detection process in case no Trojans are detected. At the beginning, since s_n is not $NULL$ and time bound has not been reached, SEL-CON-TEST-GEN is executed to explore

Algorithm 4: HARDWARE-TROJAN-DETECTOR(duv, π, tb, $golden$, $config$)

```
1  TC ← {π}
2  s_0 ← INITIALIZE(duv, π, tb),   s_n ← s_0
3  FPS ← {s_0}, SPS ← ∅                          ▷ FPS and SPS are queues
4  INTS ← GET-INTERESTED-CODE-RANGE(config)
5  β ← GET-TIME-BOUND(config)
6  while (s_n ≠ NULL) ∧ (time < β) do
7      {S', τ} ← SEL-CON-TEST-GEN(s_n, INTS)
8      if τ == NULL then
9          s_n ← STATE-SELECTOR(duv, tb, FPS, SPS, S')
10         continue
11     ret ← HT-DETECTOR(duv, golden, tb, τ)
12     if (ret) then
13         return TC, τ
14     TC ← TC ∪ {τ}
15     s_n ← STATE-SELECTOR(duv, tb, FPS, SPS, S')
16 return TC
```

the initial state (line 7). Upon completion, it returns a set of new states S' and a test case τ for s_n. If τ is $NULL$, no test case is generated for the explored state (line 8). Thus, another state is selected and explored (line 9–10). If τ is not $NULL$, HT-DETECTOR is invoked to detect hardware Trojans with the newly generated test case τ (line 11). If Trojans are detected, the algorithm returns all

generated test cases and the test case that triggers the Trojans (line 12–13). Then, HARDWARE-TROJAN-DETECTOR terminates. Otherwise, the new test case τ is added to TC, followed by invoking STATE-SELECTOR, which selects another state s_n for concolic execution (line 14–15). The variable *time* (line 6) denotes the total time elapsed since HARDWARE-TROJAN-DETECTOR starts. If there are no more states left or the time bound has been reached, HARDWARE-TROJAN-DETECTOR returns all generated test cases and terminates (line 16). We will discuss the details of each component in the following section.

5.3 HARDWARE TROJAN DETECTION

This section presents selective concolic testing for hardware Trojan detection in behavioral SystemC designs in details. We will first describe the core of our proposed approach, SCE, which includes our two primary optimizations, selective concolic test generation and coverage-guided state search strategy. Then, we will present HTD that detects hardware Trojans with the generated test cases by SCE.

5.3.1 Selective Concolic Test Generation

Traditional concolic test generation approaches generate test cases along an entire concrete execution path, as shown in Figure 2.7. However, often only a small portion of the path is from the DUV code. Most code composed of the path is from libraries, which is of no interest to users generally. It may not be beneficial to generate test cases in these libraries code for verifying the DUV. Figure 5.3 demonstrates the case. The circles denote branch points, while the arrows indicate the execution sequence. As the figure shown, although there are three paths indicated with red, blue, and magenta in the libraries, it is only one path from the

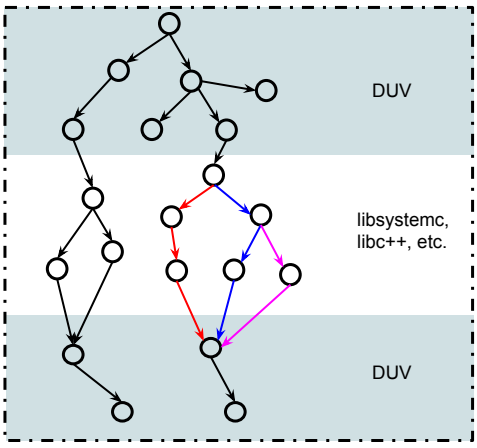

Figure 5.3: Selective concolic test generation

DUV perspective. To improve the verification efficiency, concolic test generation approaches should be restricted to generate test cases for the DUV only, by which the number of generated test cases can be reduced and hence test generation time is reduced. Furthermore, the subsequent simulation time can also be reduced with fewer test cases. Unfortunately, traditional concolic test generation approaches are not able to distinguish DUV code from libraries code. Therefore, they usually generate many redundant test cases in terms of the DUV, since these test cases follow the same path from the DUV point of view.

Our proposed selective concolic test generation approach is able to generate test cases for a specific part of code. In this case, it is the DUV. However, our approach can also be used to generate test cases for a specific library, or a combination of multiple code segments, depending on users' interests. Algorithm 5 describes our selective concolic test generation process. The procedure SEL-CON-TEST-GEN takes two parameters, s_n and $INTS$, as inputs. The input s_n is the current execution state with the concrete values for the symbolic variables that are obtained in Algorithm 6. The input $INTS$ includes all code ranges of interest to users. The

Algorithm 5: SEL-CON-TEST-GEN($s_n, INTS$)

```
1  S ← ∅, τ ← NULL
2  while HAS-NEXT-INSTRUCTION(s_n) do
3      I ← GET-NEXT-INSTRUCTION(s_n)
4      EXECUTE-INSTRUCTION(I)
5      if I is branch then
6          bp ← GET-SYMBOLIC-BRANCH-PREDICATE(I)
7          if FIND(I, INTS) then
8              s'_n ← FORK(s_n, ¬bp)
9              S ← S ∪ s'_n
10         ADD-CONSTAINTS(s_n, bp)
11     SET-NEXT-INSTRUCTION(s_n)
12 τ ← CONSTRAINT-SOLVER( GET-CONSTRAINTS(s_n) )
13 return S, τ
```

set S saves newly forked states from s_n and the test case τ will be the generated test case for s_n (line 1). If there is an instruction for execution (line 2), the instruction I is fetched (line 3) and executed (line 4). If it is a branch instruction (line 5), the symbolic predicate bp of I is computed (line 6). If I is in the ranges $INTS$, then a new state is forked with negation of bp and the state is added to the set S (line 7–9). Otherwise, no new state is forked. Afterwards, bp is added to the constraints of s_n (line 10). Then, line 11 sets the next instruction to be executed. If every instruction of s_n has been executed, a test case is generated if possible and saved to τ (line 12). Finally, the newly forked states S and the test case τ are returned.

5.3.2 Coverage-guided State Search Strategy

Algorithm 6 presents our coverage-guided state search strategy. If HT-Detector does not detect hardware Trojan, then, State-Selector is invoked to select another state for concolic test generation. The state s will be the returned state, which is $NULL$ initially (line 1). For each state s_t from S' that is forked from previous concolic execution, its path constraints are sent to a solver (line 3). If it returns $NULL$, the current state may not be reachable so that we do not save it (line 4–5). This prevents the state from being explored later, which reduces the overall execution time. If the solver succeeds, the concrete values cv are saved to the state for later execution (line 6). Then, coverage is analyzed with cv (line 7). If new code is covered, then s_t is added to the first priority state queue FPS (line 9). Otherwise, it is added to the second priority state queue SPS (line 11). After each state is evaluated, the first state in FPS is assigned to s (line 13) and is removed (line 14) if FPS is not empty. Otherwise, the first state in SPS is retrieved and removed (line 16–17). If both FPS and SPS are empty, then s remains $NULL$, which means no more state to be explored. Finally, the selected state s is returned.

5.3.3 Hardware Trojan Detection

Algorithm 7 demonstrates our hardware Trojan detection procedure. After a state is explored and the test case is generated, HT-Detector is called with four arguments, a *duv* and its golden model *golden*, a testbench *tb*, as well as the newly generated test case τ. The test case τ is simulated on both *duv* and *golden* (line 1 and line 2 respectively). If the results are not the same from both simulations, then this indicates that a Trojan is detected (line 4); otherwise, HT-Detector returns false to indicate that no Trojan is discovered.

Algorithm 6: STATE-SELECTOR(duv, tb, FPS, SPS, S')

```
1  s ← NULL
2  foreach s_t ∈ S' do
3      cv ← CONSTRAINT-SOLVER( GET-CONSTRAINTS(s_t) )
4      if cv == NULL then
5          continue
6      SET-VALUE (s_t, cv)
7      newly_covered ← COVERAGE-ANALYZER(duv, tb, cv)
8      if (newly_covered) then
9          FPS ← FPS ∪ {s_t}
10     else
11         SPS ← SPS ∪ {s_t}
12 if FPS ≠ ∅ then
13     s ← FPS.front()          ▷ get the first state from FPS
14     FPS.pop()                ▷ remove the first state from FPS
15 else if SPS ≠ ∅ then
16     s ← SPS.front()          ▷ get the first state from SPS
17     SPS.pop()                ▷ remove the first state from SPS
18 return s
```

Algorithm 7: HT-DETECTOR(duv, $golden$, tb, τ)

```
1 res1 ← SIMULATOR(duv, tb, τ)
2 res2 ← SIMULATOR(golden, tb, τ)
3 if res1 ≠ res2 then
4     return true                    ▷ indicates that Trojan is detected
5 else
6     return false
```

5.4 EXPERIMENTAL RESULTS

We have implemented the proposed approach as an automated prototype, namely SCT-HTD, based on our conclic testing of SystemC designs approach presented in Chapter 4. We have applied this approach to the open-source benchmark suite, S3CBench [98]. Although S3CBench only consists of behavioral synthesizable SystemC designs, our approach also works for non-synthesizable SystemC designs. S3CBench contains 10 SystemC designs that include multiple types of hardware Trojans based on trigger mechanism, either sequential or combinational. Half of the designs are computationally intensive designs such as image processing algorithms. Each design has fixed computation procedures for any inputs. As we all know, concolic test generation is based on branch conditions, which makes it very powerful to explore deep paths with complex conditions and corner cases. However, it is not good at generating test cases for a design that has fixed execution steps, since an execution path of such a design does not depend on input values. This is a known limitation of concolic testing. Thus, we conducted experiments on non-computationally intensive designs. The experiments were conducted on a

laptop with a 4-core Intel(R) Core(TM) i7-4700MQ CPU, 16 GB of RAM, and running the Ubuntu Linux OS with 64-bit kernel version 4.15. Table 5.1 presents our experimental results, as well as comparison with the state-of-the-art approaches. We will discuss the table in the following.

5.4.1 Effectiveness and Efficiency

We developed a golden model for each design on which we experimented. The first column of Table 5.1 gives the name (before the hyphen part) of each design. The part after the hyphen denotes the type of inserted Trojan. Details about the benchmark and the Trojans can be found in S3CBench [98]. We have evaluated the performance of SCT-HTD from three perspectives, namely the number of generated test cases to trigger the Trojan (column 2), time usage (column 6) and maximum memory usage (column 10). To pursue a fair comparison, we also set a two-hour time bound during experiments as AFL-SHT [63] did. T.O. indicates that the two-hour time bound is reached and thus the hardware Trojans are not detected. The designs that we evaluated on includes the three typical hardware Trojan types in terms of payload, namely functionality modification (`adpcm`, `fir`, and `bSort`), denial of service (`uart`), and sensitive information leakage (`aes`). As demonstrated, our approach is able to detect all three types of hardware Trojans with a few test cases, short time usage and reasonable memory usage, which demonstrates the effectiveness and efficiency of our approach.

5.4.2 Evaluation of Two Optimization Strategies

To illustrate the advantages of our selective concolic testing and coverage-guided state search strategy, we have conducted experiments with traditional concolic

Table 5.1: Experimental results of SCT-HTD and comparison with the state-of-the-art approaches

Designs	**# Test case**				**Time (s)**				**Memory (MB)**			
	SCT-HTD*	CT$^{\zeta}$	AFL$^{\eta}$	AFL-SHT$^{\delta}$	SCT-HTD	CT	AFL	AFL-SHT	SCT-HTD	CT	AFL	AFL-SHT
`adpcm-swm`	27	525	451563	423	157	T.O.	T.O.	1.71	3546	14337	N/A	N/A
`adpcm-swom`	7	503	450839	414	31	T.O.	T.O.	1.67	1341	14442	N/A	N/A
`fir-cwom`	26	76	207	41	13	26	8.51	0.07	1621	2305	N/A	N/A
`bSort-cwom`	2	2	118	39	8	10	4.82	0.05	1074	2668	N/A	N/A
`bSort-swm`	4	975	19826	108	10	T.O.	337.36	0.11	1106	10768	N/A	N/A
`uart-swm1`	3	1023	N/A	N/A	9	T.O.	N/A	N/A	1071	13011	N/A	N/A
`uart-swm2`	3	1016	172	51	9	T.O.	8.82	0.18	1070	12972	N/A	N/A
`aes-cwom`	11	11	50544	22	23	32	888.29	0.04	1386	1396	N/A	N/A

* Our approach $^{\zeta}$ Concolic testing (CT) without our optimizations $^{\eta}$ Fuzzing with software-oriented mutation

$^{\delta}$ Coverage-guided fuzzing

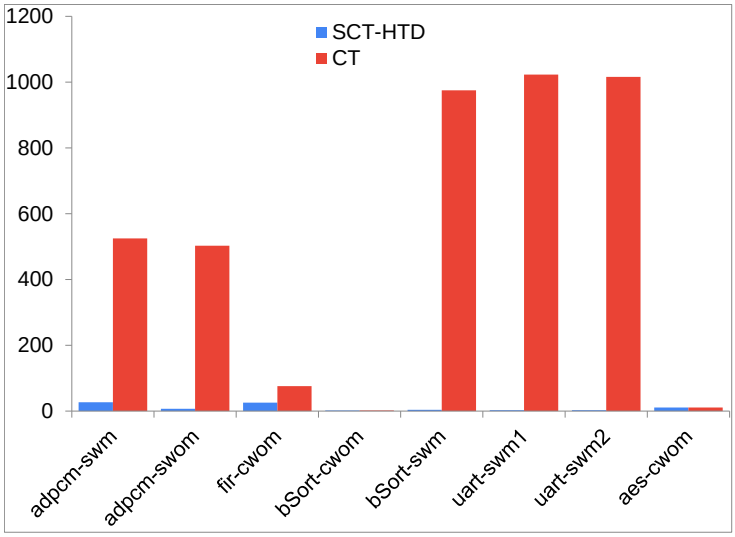

Figure 5.4: Number of generated test cases

testing approach (without the two optimization), denoted as CT. Column 3, column 7 and column 11 show the number of generated test cases, time usage and maximum memory usage with CT, respectively. Five out of eight Trojans are not detected within the two-hour time bound, although many test cases are generated. Figure 5.4, Figure 5.5, and Figure 5.6 demonstrate the advantage of SCT-HTD in the three aspects graphically, compared with traditional concolic testing approach. As shown in figures, our optimization strategies reduce the number of generated test cases, time usage, and memory usage tremendously for more than half of the designs. For other designs, our approach is also able to detect hardware Trojans with fewer or equal number of generated test cases, less time and memory usage.

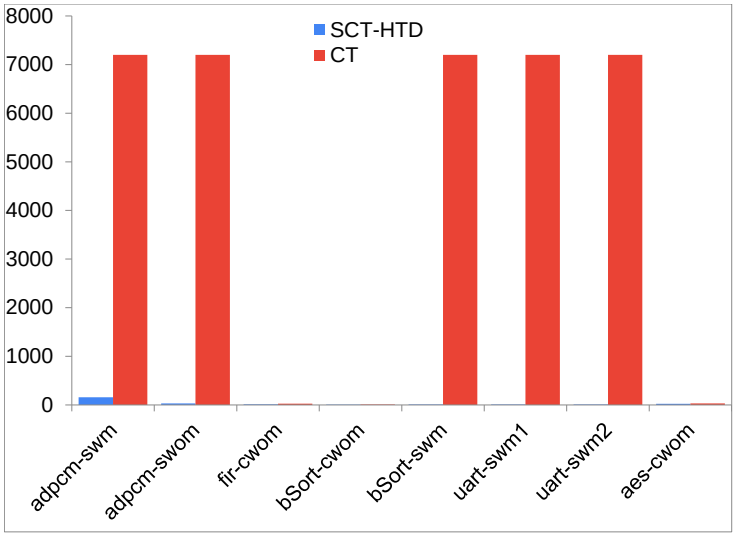

Figure 5.5: Time usage

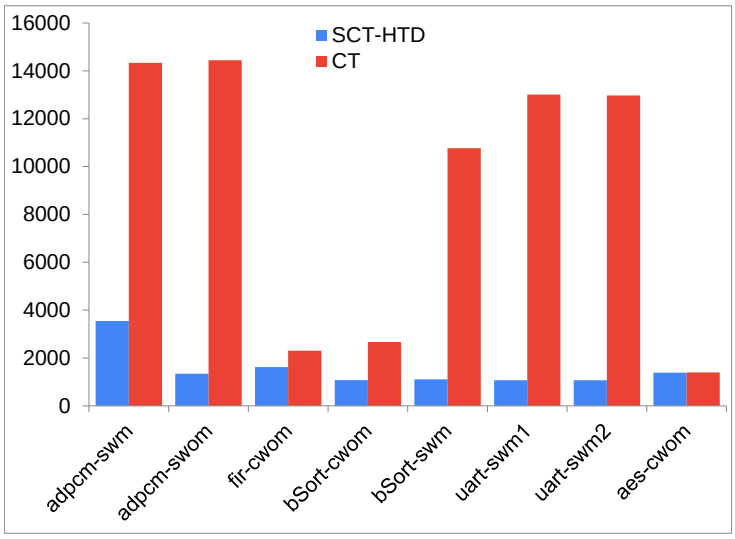

Figure 5.6: Maximum memory usage

5.4.3 Comparison with State-of-the-Art Approaches

Two existing approaches target hardware Trojan detection for behavioral SystemC designs. One [97] uses property checking and the other, AFL-SHT [63], adopts coverage-guided fuzzing. Although we do not have access to the commercial formal tools to conduct experiments, formal approaches on Trojan detection for behavioral SystemC designs are not as promising as AFL-SHT. Since the source code of AFL-SHT is not available, we take the results obtained by both AFL-SHT and AFL [101] from the paper for comparison. Some data are not presented in the paper, such as memory usage, which are denoted as N/A. As illustrated, our approach is able to detect the Trojans with far fewer test cases than AFL and AFL-SHT. Time usage of SCT-HTD is much less than AFL for half of the designs. For other designs, SCT-HTD uses a little longer time than AFL and AFL-SHT. There are two possible reasons. First, power of machines running experiments are different. We conducted experiments on a laptop which might be less powerful. Second, concolic testing involves constraint solvers to generate test cases, which is a known time-consuming operation compared with fuzzing techniques. However, the increase of time usage is moderate.

5.5 SUMMARY

In this chapter, we have presented a novel approach for detecting hardware Trojans in behavioral SystemC designs with selective concolic testing. We have also proposed an algorithm to improve the efficiency of our approach with coverage-guided state search strategy. We have implemented the proposed approach as a prototype, namely SCT-HTD. To show the effectiveness and efficiency of the proposed approach, we have conducted experiments on an open source benchmark

that includes multiple types of hardware Trojans based on trigger mechanisms. The results demonstrate that our approach is very promising on hardware Trojan detection for behavioral SystemC designs.

Chapter 6

SCBENCH BENCHMARK

6.1 MOTIVATION

SystemC verification has been studied for around two decades. It is both necessary and important for verification researchers to evaluate their new approaches and algorithms using common and updated benchmark suites, no matter which paradigm they adopt. However, so far, different verification approaches are evaluated on different sets of SystemC designs, among which some designs are not updated according to the latest SystemC Standard [49]. Lacking common benchmarks makes it difficult to compare the performances of various approaches. With common benchmark suites, researchers are able to compare the effectiveness and efficiency of their approaches with the state-of-the-art techniques. The common benchmarks should cover as many application domains and SystemC core features as possible. It also should conform to the latest SystemC Standard.

Previously, several sets of SystemC designs have been developed and used by various researchers. KRATOS [17] provides a set of SystemC designs for its safety property checking, which are not updated according to the latest SystemC Standard. SCIVER [36] provides a few SystemC transaction level modelling designs for its high-level property checking approach. S2CBench [86] consists of a set of high-level synthesizable SystemC designs, which is a subset of SystemC. S2CBench is mainly targeted for evaluating high-level synthesis tools. SEQ [92] provides a

set of SystemC designs in both low level and high level. They are mainly intended for equivalence verification and cover only a few application domains. Designs in Aegis [1] model some classical concurrency scenarios, which are mainly used to evaluate approaches for analyzing concurrency related errors. All the aforementioned benchmarks either cover a small subset of SystemC core features, or target a specific purpose.

In this chapter, we present SCBench, a comprehensive suite of benchmark designs for SystemC verification and validation. SCBench can be used for various purposes, such as formal verification, dynamic validation, and data race detection [68]. Key features of SCBench are summarized as follows.

- SCBench consists of 38 well-written representative SystemC designs that cover a variety of application domains, such as CPU architecture, security, digital signal processing (DSP), networking, and artificial intelligence (AI).
- The designs range from small single process designs to large multi-process designs. All designs are selected carefully to cover as many SystemC core features as possible. Five out of 38 designs are modelled specifically following TLM-2.0 [49].
- Each design has been provided a set of stimuli and a testbench including stimuli applications and output monitors.
- SCBench is freely available online to all researchers [85].

6.2 OVERVIEW

The goal of the SCbench benchmark suite is to promote research on SystemC verification by providing a set of representative designs that are reasonably large and

well-written. We do not intend to evaluate the performances of existing SystemC verification approaches. Therefore, all designs follow the latest SystemC Standard, but not specific restrictions of certain verification approaches.

The SCBench benchmark suite comprises 38 designs that have been selected from various application domains, such as CPU architectures, security, DSP, image processing, networking, and AI. Each domain offers different design characteristics. All designs are selected carefully to cover a wide variety of SystemC core features, such as hierarchical structures, hardware-oriented data types, and bit-precise operations.

Table 6.1 shows the summary of the SCBench benchmark suite. The first three columns present the names of the designs, the number of processes, and LoC, respectively. LoC is calculated using `cloc`. Note that only the code in a design itself is taken into account, excluding the testbench code. Thus, the actual sizes of the source files are larger. The last column indicates the sources [1,5,27,41,60,86,92,95] of every design.

Table 6.1: Summary of SCBench benchmark suite

Designs	**# of Proc.**	**LoC**	**Type**	**Source**
RISC CPU	13	2056	CI/CD	SEQ [92]
RISC_CPU_control	1	826	CD	SEQ
RISC_CPU_mmxu	1	193	CI	SEQ
RISC_CPU_exec	1	128	CI	SEQ
RISC_CPU_floating	1	127	CI	SEQ
IA-32	1	336	CI/CD	SEQ

Table 6.1: Summary of SCBench benchmark suite (continued)

Designs	**# of Proc.**	**LoC**	**Type**	**Source**
MIPS	1	257	CI/CD	SCBench
Y86	11	301	CI/CD	SEQ
AES	31	1624	CI	Scoot [5]
DES	14	2401	CI	Scoot
RSA	1	324	CI	SystemC library [95]
KASUMI	2	415	CI	S2CBench [86]
SNOW3G	1	522	CI	S2CBench
MD5C	1	271	CI	S2CBench
IDCT	1	450	CI	S2CBench
Interpolation	1	231	CI	S2CBench
ADPCM	1	270	CD	S2CBench
FFT	1	334	CI	S2CBench
ASR/ABS	1	249	CI/CD	STATE [41]
UsbTxArbiter	5	144	CD	SEQ
UART	1	127	CI	S2CBench
Qsort	1	204	CI	S2CBench
Disparity	4	634	CI	S2CBench

Table 6.1: Summary of SCBench benchmark suite (continued)

Designs	**# of Proc.**	**LoC**	**Type**	**Source**
Sobel	1	269	CI	S2CBench
VGA	2	218	CD	S2CBench
NoC	656	2130	CI/CD	opencores.org [60]
Master/Slave Bus	5	974	CD	SystemC library
Pkt_switch	17	376	CD	SystemC library
ANN	2	315	CI	S2CBench
Crossroad	4	74	CD	Aegis [1]
Philosophers	10	116	CD	Aegis
Producer/Consumer	2	44	CD	Aegis
SimpleRing	6	211	CD	Aegis
TLM_b_transport	2	102	CD	www.doulos.com [27]
TLM_DMI_DBG	2	202	CD	www.doulos.com
TLM_routing	6	277	CD	www.doulos.com
TLM_nb_transport	2	390	CD	www.doulos.com
AMBA_AHB	7	1542	CI/CD	STATE

6.3 DESIGN DESCRIPTIONS

The systems or algorithms implemented by these designs are widely used in the real world. Their functional descriptions are described as follows, categorized by application domains (AD). Note that some designs are relevant to multiple categories. However, each of them is put into one category for organization purposes. In addition, the designs are also classified control-dominant (CD) or computation-intensive (CI), as shown in the fourth column of Table 6.1.

AD1: CPU Architectures. This category consists of designs that model multiple CPU architectures including RISC CPU, Intel's IA-32, and MIPS architectures. These designs can be used as instruction set simulators that software developers can use to test their software in the early stage of development.

`RISC CPU` models a CPU architecture that fetches instructions, decodes and executes them, and then writes the results back to registers or memory. It provides more than 39 instructions including arithmetic, logical, branch, floating point, and SIMD (MMX-like). Four components of the CPU architecture, `RISC_CPU_control`, `RISC_CPU_mmxu`, `RISC_CPU_exec`, and `RISC_CPU_floating`, are listed as separate designs. They describe instruction decode unit, MMX-like execution unit, integer execution unit, and floating point execution unit, respectively.

`IA-32` is an instruction length decoder for Intel's IA-32 instruction set architecture.

`MIPS` is a simplified MIPS processor that has 30 instructions. We developed this design based on its C version [40].

`Y86` is a simple CISC CPU implementing a subset of the instructions of the IA-32 architecture. The design has nine total instructions and nine registers with 32-bit data width each.

AD2: Security. The importance of data security has increased drastically in the past decade. Therefore, we selected several algorithms for data encryption and decryption in this category.

`AES` is an encryption algorithm implementing the advanced encryption standard. It is a symmetric key algorithm in that the same key is used for both encryption and decryption.

`DES` is also a symmetric key algorithm used for encryption. It highly influenced the advancement of modern cryptography.

`RSA` implements the RSA public-key cipher that is an asymmetric cryptographic algorithm. This implementation illustrates the usage of arbitrary precision types of SystemC.

`KASUMI` is a block cipher algorithm that is used in mobile communication systems. The algorithm works with 128-bit key and 64-bit input and output.

`SNOW3G` implements the SNOW 3G algorithm, a stream cipher. It consists of two interactive components, a linear feedback shift register and a finite state machine.

`MD5C` is a widely used algorithm to generate hash values. It can also be used as a checksum to verify data integrity.

AD3: Digital Signal Processing. With the explosive growth of smart and portable devices, some DSP functionalities have been integrated into these devices. This category includes four DSP algorithms that are widely used in the field.

`IDCT` is the inverse discrete cosine transform that expresses a finite sequence of data points in terms of a sum of cosine functions of different frequencies. It is important to many applications in science and engineering.

`Interpolation` is an algorithm used to construct new data points within the

range of a discrete set of known data points. The design implements a 4-stage interpolation filter.

`ADPCM` describes adaptive differential pulse-code modulation, which is a variant of differential pulse-code modulation. The design accepts 16-bit pulse-code modulation samples as inputs and converts them into 4-bit samples.

`FFT` implements the fast Fourier transform algorithm that computes the discrete Fourier transform of a sequence. `FFT` is widely used in engineering, science, and mathematics.

AD4: Automotive and Industrial. Designs in this category are usually used in embedded systems for basic math manipulation and control systems. Typical applications include communications, performance monitors, and sensor systems.

`ASR/ABS` is an anti-slip regulation and anti-lock braking system that monitors the speed of each wheel and regulates the brake pressure to prevent loss of traction or wheel lockup. It comprises wheel speed sensors, a hydraulic modulator, an electronic control unit, and a control area network bus.

`UsbTxArbiter` describes an algorithm for processing data using an arbiter in a USBHostSlave core.

`UART` models a universal asynchronous receiver/transmitter. It translates data between characters in a computer and an asynchronous serial communication format.

`Qsort` implements the well-known quick sort algorithm that sorts data in ascending order. Sorting of information is critical in various disciplines so that priorities can be made.

AD5: Image Processing. With the great success of augmented reality and virtual reality, image and video processing have become more and more important.

Therefore, image processing is given as a separate category.

`Disparity` computes the disparity in a 3D-image. Disparity refers to the difference in image location of an object seen by the left and right eyes.

`Sobel` is an edge detection algorithm used in image processing and computer vision. The algorithm takes an image as input and returns a new image composed of the edges of the original image.

`VGA` implements the VGA controller and image generator. The controller handles the low-level details of communication with a monitor over a VGA connector.

AD6: Networking. The networking category represents designs of network devices. These designs include a network-on-chip simulator, a simple bus architecture, and a packet switch.

`NoC` is a network-on-chip simulator that supports a maximum of 4 by 4 tiles. Each tile includes an IP core, a router, and six FIFOs. The design uses the synchronizing FIFO and wormhole routing.

`Master/Slave Bus` describes a bus structure that includes a set of masters, a set of slaves, a shared bus, and an arbiter. There are three types of master interfaces: (1) blocking interface, where calls return after transmission is finished; (2) non-blocking interface, where calls return immediately; and (3) direct interface, where direct access to slaves are enabled.

`Pkt_switch`, a packet switch design, implements a 4x4 multicast helix packet switch. The switch uses a self-routing ring of shift registers to transfer cells from one port to another in a pipelined style. Input and output ports have FIFO buffers of depth four each.

AD7: Artificial Intelligence. The category includes a widely used algorithm,

artificial neural network (ANN), in machine learning and cognitive science. The network is a computational model based on a large collection of neural units modelling the way the brain solves problems. It consists of multiple layers.

Besides the specific seven categories of designs described previously, the benchmark also includes four designs from Aegis project [1] that model classical concurrency scenarios, such as dining philosophers and producer/consumer problems. These designs are mainly intended to evaluate approaches for analyzing concurrency related errors, such as race conditions. Particularly, Crossroad and Philosophers intentionally include race conditions.

In addition, the benchmark contains five designs that are modelled explicitly following TLM-2.0 Standard. TLM_b_transport is a simple loosely-timed model with blocking transport interface in TLM-2.0 style. The design consists of two modules, one initiator and one target. The target module represents simple memory. The initiator module generates transactions which read from or write to the memory.

TLM_DMI_DBG explores the response status of the generic payload, as well as the direct memory and the debug transport interfaces. The purpose of the direct memory interface is to speed up simulation by giving initiators a direct pointer to an area of memory in a target, by which there is no need to go through the transport interface for every read and write transaction. The debug transport interface is intended to give an initiator the ability to read or write memory in a target without causing side effects and without advancing simulation time.

TLM_routing models a router as a TLM-2.0 interconnect component, through which transactions propagate from one initiator to one of four targets. The router has a forward path that forwards transactions to the target and a return path that

returns transactions to the initiator.

`TLM_nb_transport` is an approximately-timed model with non-blocking transport interface in TLM-2.0 style. It also includes the following features of TLM-2.0: generic payload, payload event queues, memory management, the BEGIN_REQ and BEGIN_RESP exclusion rules.

`AMBA_AHB` is a real world design that implements the advanced high performance bus (AHB) of the advanced microcontroller bus architecture (AMBA). The AMBA AHB, which is currently used in many high performance SoCs, splits transactions into AMBA conforming transfers. The AMBA AHB is a synchronous clocked bus.

6.4 DESIGN ANALYSIS

This section provides detailed characteristics of the SCBench benchmark suite. Table 6.2 shows the numbers of different operations including arithmetic, comparison or relational, bitwise, and logical. Table 6.3 illustrates the numbers of various statements, such as *if, switch, while, for*, and *assignment* statements including compound assignments.

Table 6.2: Numbers of operations for each design

Designs	**Add/Sub**	**Mul**	**Div/Mod**	**Comp**	**Bitwise**	**Logic**
RISC CPU	41	6	3	92	95	5
RISC_CPU_control	9	0	0	33	0	0
RISC_CPU_mmxu	18	4	0	19	64	0
RISC_CPU_exec	6	1	2	3	9	0
RISC_CPU_floating	4	1	1	2	22	0

Table 6.2: Numbers of operations for each design (continued)

Designs	**Add/Sub**	**Mul**	**Div/Mod**	**Comp**	**Bitwise**	**Logic**
IA-32	45	4	0	45	0	7
MIPS	15	2	0	10	41	0
Y86	12	0	0	15	1	14
AES	7	0	0	15	235	30
DES	1	0	0	0	7	5
RSA	17	8	14	36	1	7
KASUMI	44	0	0	22	60	0
SNOW3G	11	0	0	10	114	0
MD5C	284	4	0	16	16	274
IDCT	123	33	0	36	31	12
Interpolation	14	10	0	8	0	2
ADPCM	15	2	0	16	6	0
FFT	17	5	2	10	0	0
ASR/ABS	9	6	2	22	0	0
UsbTxArbiter	0	0	0	5	0	0
UART	3	0	0	11	0	13
Qsort	8	0	0	0	0	2

Table 6.2: Numbers of operations for each design (continued)

Designs	**Add/Sub**	**Mul**	**Div/Mod**	**Comp**	**Bitwise**	**Logic**
Disparity	33	2	13	42	0	13
Sobel	26	2	0	17	0	0
VGA	10	0	0	13	0	6
NoC	66	0	4	10	110	20
Master/Slave Bus	27	2	13	46	0	13
Pkt_switch	13	1	1	14	12	39
ANN	7	1	0	2	5	1
Crossroad	5	0	4	3	0	1
Philosophers	5	0	5	0	0	0
Producer/Consumer	1	0	0	1	0	0
SimpleRing	19	0	9	6	0	0
TLM_b_transport	0	0	2	8	2	3
TLM_DMI_DBG	2	3	4	11	2	1
TLM_routing	8	1	7	13	8	3
TLM_nb_transport	6	1	6	22	0	6
AMBA_AHB	25	4	10	132	5	20

Note: Add — Addition, Sub — Subtraction, Mul — Multiplication, Div — Division, Mod — Modulo, Comp — Comparison.

Table 6.3: Numbers of statements for each design

Designs	**if**	**switch**	**while**	**for**	**assignment**
RISC CPU	51	6	52	1	293
RISC_CPU_control	6	3	30	0	102
RISC_CPU_mmxu	19	1	1	0	85
RISC_CPU_exec	3	1	1	0	19
RISC_CPU_floating	1	1	2	0	30
IA-32	39	1	2	9	33
MIPS	4	3	1	3	57
Y86	20	0	6	6	33
AES	26	8	1	0	246
DES	6	11	1	0	426
RSA	16	0	2	5	44
KASUMI	2	0	2	12	62
SNOW3G	1	0	2	4	109
MD5C	3	0	2	8	62
IDCT	2	0	1	3	109
Interpolation	0	0	1	5	15
ADPCM	12	0	1	1	33
FFT	2	0	10	0	60
ASR/ABS	21	1	6	3	64

Table 6.3: Numbers of statements for each design (continued)

Designs	**if**	**switch**	**while**	**for**	**assignment**
UsbTxArbiter	7	1	0	0	5
UART	11	0	2	0	37
Qsort	1	0	1	5	17
Disparity	16	0	9	11	96
Sobel	8	0	1	8	22
VGA	9	0	2	0	4
NoC	130	1	13	7	272
Master/Slave Bus	59	1	6	13	77
Pkt_switch	32	0	2	0	76
ANN	5	0	0	14	31
Crossroad	0	1	3	0	5
Philosophers	0	0	3	1	5
Producer/Consumer	1	0	2	0	4
SimpleRing	8	0	7	5	20
TLM_b_transport	5	0	0	2	5
TLM_DMI_DBG	14	0	0	4	8
TLM_routing	14	0	0	7	12
TLM_nb_transport	28	1	0	1	40
AMBA_AHB	75	13	1	9	168

Figure 6.1 and Figure 6.2, which depict the occurrence rates of operations and statements in the designs respectively, desmonstrate the more detailed characteristics of the designs visually[1]. Figure 6.1 shows that each application domain of SCBench has different characteristics in terms of operations. For example, the designs in the CPU architecture and the security domains have a higher proportion of bitwise operations compared with other designs. Figure 6.2 shows that each application domain also has different characteristics in terms of statements. Except assignments, which is dominant in every design, the designs in the DSP domain have a high proportion of loops, since they usually process a set of data repeatedly. In contrast, the designs in the network domain have a high proportion of *if* statements. This is because decision-making dominates the network devices. Thus, researchers may select specific categories for their verification purposes.

Table 6.4 presents representative data types and features of SystemC that are covered by each design, as shown in the second column. We do not list those features contained by most designs, such as signals and input/output ports. As illustrated, the designs cover a wide variety of SystemC hardware-oriented data types and core features, such as fixed and arbitrary precision integral types, fixed point types, FIFOs and signals. We summarize the core features of SystemC language including the TLM-2.0 Standard in three categories, as shown in the first two columns of Table 6.5. The last column indicates whether or not each feature is covered by our benchmark suite, denoted as C (Covered) or N (Not covered), respectively. As can be seen, 22 out of 29 core features of SystemC language are covered by our benchmark suite.

[1]The occurrence rate of each operation is the proportion of the number of the operation to the total number of operations counted. So is the occurrence rate of each statement.

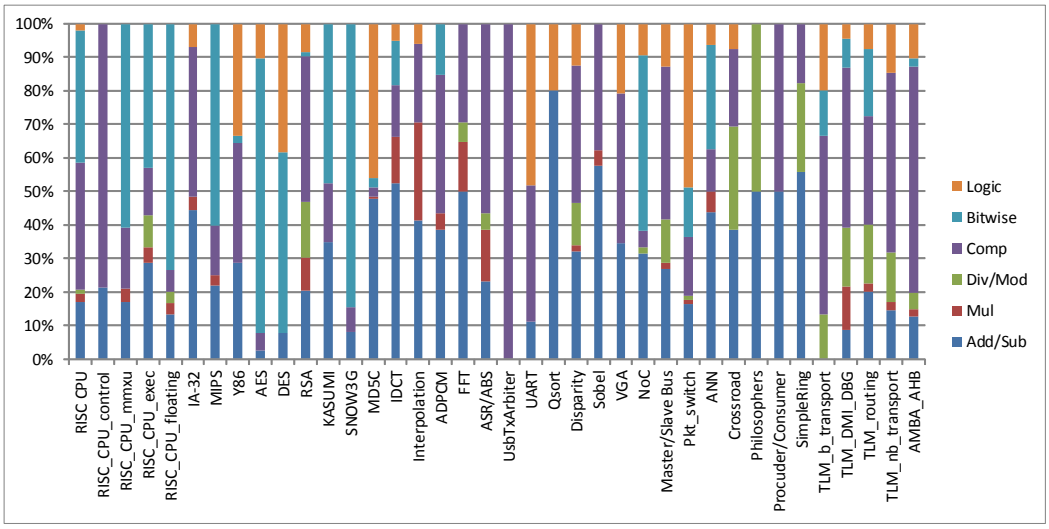

Figure 6.1: Occurrence rates of operations per design

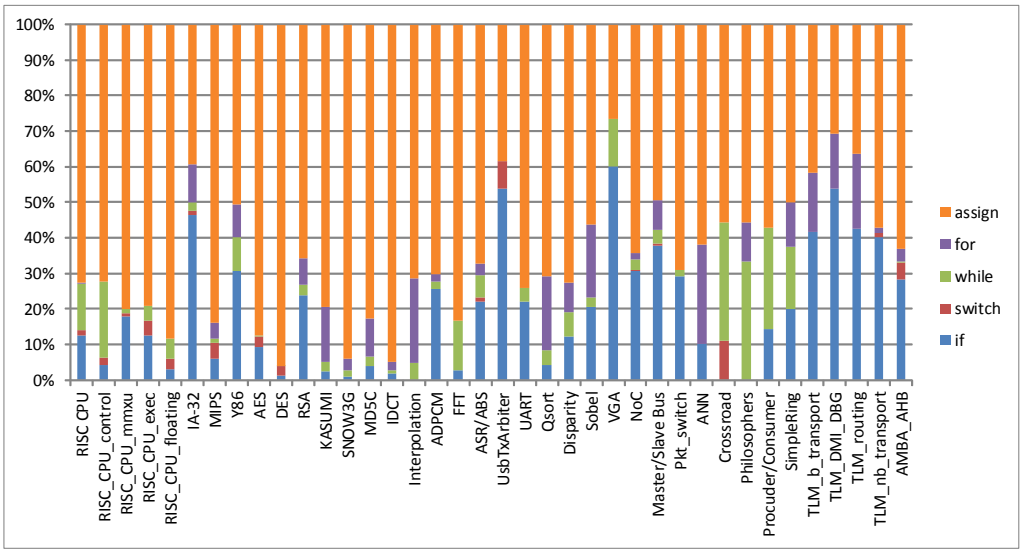

Figure 6.2: Occurrence rates of statements per design

Table 6.4: Representative data types and features of SystemC

Designs	**Representative data types and features**
RISC CPU	Fixed precision integer, bit manipulation, floating point, multi-process, hierarchical design
RISC_CPU_control	Fixed precision integer, clocked thread process
RISC_CPU_mmxu	Bit manipulation, clocked thread process
RISC_CPU_exec	Clocked thread process
RISC_CPU_floating	Floating point, bit manipulation
IA-32	Three-valued logic
MIPS	Bit manipulation
Y86	Fixed precision integer, multi-process
AES	Arbitrary precision integer, bit manipulation, multi-process
DES	Fixed precision integer, multi-process
RSA	Arbitrary precision integer
KASUMI	Two-dimension array of port, large fixed arrary
SNOW3G	Array of port, large fixed array, bit manipulation
MD5C	Arrays of different bit widths, bit manipulation
IDCT	Fixed precision integer, bit manipulation
Interpolation	Fixed point data, polynomial decomposition
ADPCM	Fixed precision integer
FFT	Fixed point data
ASR/ABS	FIFO
UsbTxArbiter	Multi-process

Table 6.4: Representative data types and features of SystemC (continued)

Designs	**Representative data types and features**
UART	Clocked thread process
Qsort	Pointer
Disparity	Fixed precision integer, hierarchical design
Sobel	FIFO, arbitrary precision integer, interface, bit manipulation, hierarchical design
VGA	Clocked thread process
NoC	FIFO, fixed precision integer, interface, hierarchical design
Master/Slave Bus	Interface, multi-process, hierarchical design
Pkt_switch	Fixed precision integer, multi-process
ANN	Array of port
Crossroad	Multi-process, synchronization
Philosophers	Multi-process, synchronization
Producer/Consumer	Multi-process, synchronization
SimpleRing	Multi-process, synchronization
TLM_b_transport	Blocking transport interface, LT coding style
TLM_DMI_DBG	Direct memory interface, debug transport interface
TLM_routing	Interconnect component
TLM_nb_transport	Non-blocking transport interface, AT coding style, payload event queues
AMBA_AHB	Non-blocking transport, generic payload, arbiter

Table 6.5: Coverage of SystemC core features

	SystemC Specifics	**Covered/Not covered**
Core language	Modules	C
	Hierarchical modules	C
	SC_METHOD	C
	SC_THREAD	C
	SC_CTHREAD	C
	Ports	C
	Events	C
	Interfaces	C
	Timers	C
	Signals	C
	FIFOs	C
	Mutexes	N
	Semaphores	N
Data types	Bit vectors	C
	Fixed-point numbers	C
	Fixed-precision integral types	C
	Arbitrary-precision integral types	C
	Four-valued logic types	N
	Logic vectors	N
	Resolved types	N
TLM-2.0	LT coding style	C
	AT coding style	C
	Blocking transport interface	C
	Non-blocking transport interface	C
	Direct memory interface	C
	Debug transport interface	C
	Generic payload	C
	Global quantum	N
	Combined interfaces	N

Note: C — Covered, N — Not covered.

6.5 DESIGN VALIDATION

We provide a testbench for each design to verify its functionality. Figure 6.3 shows the architecture of a testbench of a SystemC design. The testbench consists of a `Driver` part and a `Monitor` part. The `Driver` part sends the stimuli to a DUV, while the `Monitor` part collects the output responses and compares them with the golden outputs. In addition, the testbench includes an option to dump a VCD file to view the waveforms of signals.

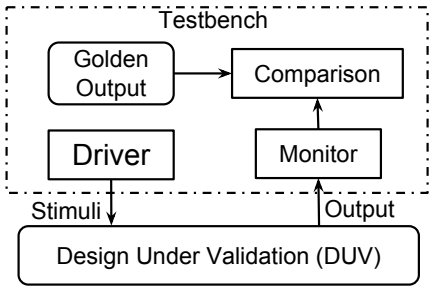

Figure 6.3: Testbench for a SystemC design

For each design, we also provide a set of stimuli, which can be modified by users. The benchmark is validated with the given stimuli on a desktop with a 4-core Intel(R) Core(TM) i7-4790 CPU, 16 GB of RAM, and running the Ubuntu Linux OS with 64-bit kernel version 3.19. It can be migrated to other OS easily. We computed the code coverage reported by `LCOV`, as shown in Figure 6.4. The code coverage can serve as a baseline for future verification projects.

6.6 SUMMARY

SystemC verification and validation requires high quality benchmarks to evaluate new approaches and compare them with the state-of-the-art approaches. We have presented such a benchmark in this chapter, namely SCBench, which complies with

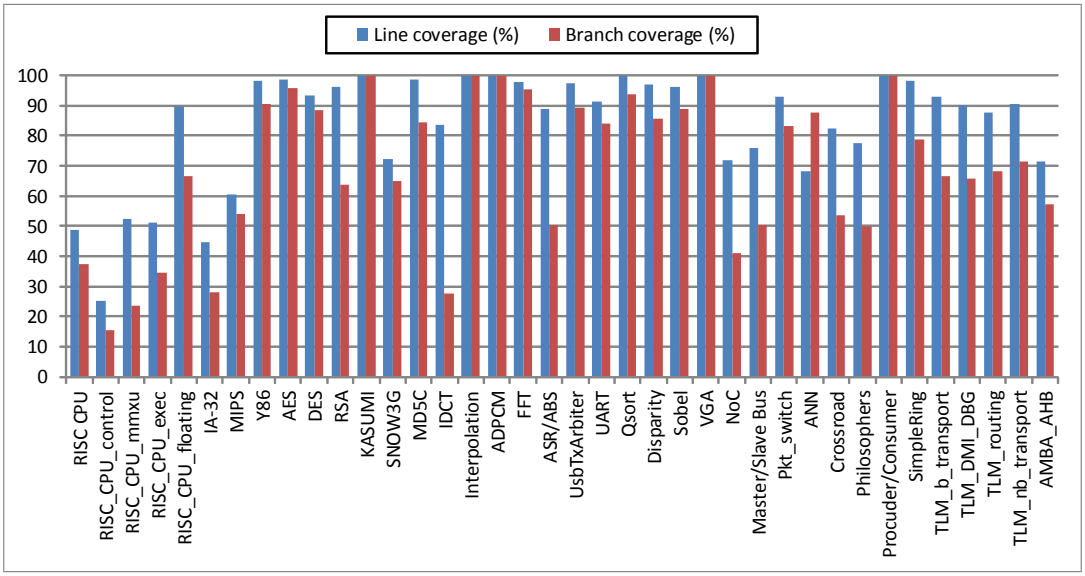

Figure 6.4: Code coverage results of the benchmark suite

the latest SystemC Standard. The SCBench benchmark suite contains 38 representative SystemC designs chosen carefully to cover as many application domains and SystemC core features as possible. SCBench can be used for various purposes, such as formal verification, dynamic validation, data race detection, and others. For example, the 13 designs from S2CBench except `FFT` are high-level synthesizable so that they can be used to evaluate the performance of high-level synthesis tools in addition to verification and validation.

We have also provided a testbench and a set of stimuli for each design. Moreover, SCBench is freely available to all researchers [85]. They can apply their approaches to these SystemC designs directly, or adapt the designs to their specific purposes.

Chapter 7

RELATED WORK

7.1 SYSTEMC VERIFICATION

SystemC verification has been studied for around two decades. The existing approaches to SystemC verification can be largely divided into two paradigms: *formal verification* and *simulation-based verification*, also known as dynamic validation. There are also a handful of hybrid approaches that combine formal and simulation-based techniques, as well as other emerging techniques for SystemC verification [69].

7.1.1 Formal Verification of SystemC Designs

There have been many attempts to formally verify SystemC designs by property checking. Table 7.1 presents the summary, including adopted techniques and primary limitations of formal approaches to SystemC verification in the literature. In general, formal verification of SystemC designs currently has the following limitations. First, model checkers accept models that are represented in specific verification languages, not the SystemC language directly. Therefore, all those formal approaches translate SystemC designs, as well as defined properties, into specific representations accepted by certain model checkers, most of which are manually translated. In addition, the simulation scheduler of SystemC also has to be modeled in the specific representation. The translation itself is error-prone and

time-consuming. Second, property formulation is challenging. For each SystemC design, a set of properties has to be defined first. The verification quality of a design highly depends on the quality of the properties. Last but not least, all existing formal approaches to SystemC verification are not scalable based on their experiments. No formal approach yet shows its applicability to real industrial scale SystemC designs.

Table 7.1: Summary of formal approaches for SystemC verification

References	**Techniques**	**Primary limitations**
Kroening and Sharygina [56]	Model checking	Not scalable (case study on a very simple design and only checks one property).
Karlsson *et al.* [53]	Model checking	Two-step translation; explicitly model the simulation kernel in PRES+; limited support for SystemC features.
Herber *et al.* [41]	Model checking	Not scalable (case study on two small designs); focused on translation from SystemC to UPPAAL timed automata but no tailored approach to model checking of SystemC designs.
Chou *et al.* [14]	Invariant checking, BMC, Simbolic simulation	Customized simulation kernel; proves deadlock properties up to a certain bound; limited support for SystemC features.
SCIVER [36]	BMC, induction	Proves properties up to a certain bound; only applicable to untimed SystemC designs; no advanced techniques to reduce redundant scheduling sequences.

Table 7.1: Summary of formal approaches for SystemC verification (continued)

References	**Techniques**	**Primary limitations**
KRATOS [17–20]	BMC, POR	Slow abstraction refinement by the formal engine; very limited support of underlying translator from SystemC to C. For example, pointers and arrays are not supported yet.
Pockrandt *et al.* [82]	Model checking	Inaccurate modeling of TLM interfaces.
SDSS [15, 16]	BMC, POR, induction	Cannot handle cyclic states; proves safety properties up to a certain bound.
SISSI [64]	Symbolic simulation, POR	Only applicable to designs that either terminate or contain bugs; cannot detect loops; no advanced techniques for alleviating path explosion.
VERDS [102]	SymMC, POR	Manual formal modeling; limited support for SystemC by guarded assignment systems; not scalable.
Herber and Hünnemeyer [44]	Model checking	Many assumptions such as no integer overflow, proper memory access of pointers; No hardware data types and inheritance support, which is one of the cores of SystemC.
ESS [46]	Symbolic simulation	Not scalable due to the limited support for SystemC features by IVL; no advanced techniques for alleviating path explosion.

Table 7.1: Summary of formal approaches for SystemC verification (continued)

References	**Techniques**	**Primary limitations**
CSS [47]	Symbolic simulation	Not scalable due to the limited support for SystemC features by XIVL.
Hajisheykhi *et al.* [39]	Model slicing, model checking	Combined multiple existing tools; conducted experiments on very simple designs.
Liebrenz *et al.* [65]	Model checking	Limited support for SystemC features.
Veeranna and Schafer [97]	Model checking	Highly dependent on stimuli provided by IP vendors; detects specific types of Trojans.

7.1.2 Simulation-based Verification of SystemC Designs

There are quite a few of simulation-based approaches to SystemC verification in the literature as well. Table 7.2 presents the summary of existing simulation-based approaches to SystemC verification, including adopted techniques and primary limitations. Generally, simulation-based approaches can be classified into two categories, test generation and runtime monitoring. Runtime monitoring mainly adopts assertion-based verification, while test generation uses various techniques. The primary limitation of most existing simulation-based approaches is the copious amount of manual intervention that is necessary. Examples of such intervention include manual instrumentation and mutants and assertions development. Additionally, performance overheads for runtime monitoring approaches are noticeable.

Table 7.2: Summary of simulation-based approaches for SystemC verification

References	**Techniques**	**Primary limitations**
SCV library [95]	random testing	Not effective.
Ferrandi *et al.* [32]	Constraint solving	Manually generation of finite state machines; the semantics of the SystemC simulator is not considered.
Bruschi *et al.* [7]	Constraint solving	Works only for synchronous designs.
Junior *et al.* [51]	Code-coverage analysis	Manual instrumentation.
Ecker *et al.* [30]	Assertion-based verification	Proof-of-concept; runtime increases.
Pierre and Ferro [80]	Assertion-based verification	Modification of SystemC designs.
Kallel *et al.* [52]	Aspect-oriented programming	Runtime increases; customized aspect and monitor for each design; verification results highly depend on simulation inputs.
Sen [88] [87]	Mutation testing	Mutant development is labor intensive; exercising mutants is time consuming.
Kuznik and Müller [59]	Functional coverage library	No performance evaluation; no usage example.
ARTEST [11] [12]	SMV	SystemC design translation; test case refinement.
CHIMP [28] [29]	Assertion-based verification	Time-consuming and laber-intensive to develop temporal assertions manually; Customized SystemC library.
HRD [89]	lockset, happens-before	High simulation overhead; false alarms.

7.1.3 Hybrid Approaches to SystemC Verification

Formal approaches intend to verify the correctness of a DUV, usually by property checking with mathematical guarantees. However, property formulation is very challenging for a DUV, especially for SystemC designs. This is due to the complexity of the SystemC language. In addition, state-space explosions limit the scalability of formal techniques. Simulation-based approaches usually is unable to simulate all possible input combinations for a DUV. Each technique has its own advantages and disadvantages. Thus, researchers have combined these two techniques together to overcome the shortcomings of each other.

Table 7.3 presents the summary, including adopted techniques and primary limitations of the aforementioned hybrid approaches to SystemC verification. The hybrid approaches generally combine simulation, formal techniques, or other static analysis techniques. Most current hybrid approaches simulate SystemC designs first and then model checking is employed based on the simulation. Thus, verification results by model checking highly depend on simulation that requires test inputs. However, most approaches utilize random test generation, which may compromise the verification results.

7.1.4 Emerging Techniques for SystemC Verification

Machine learning is a powerful technique, which has been successfully used in many areas in recent years. It has also been explored in the EDA field [94, 99, 100]. Recently, machine learning has also been studied for SystemC verification. Efendioglu *et al.* [31] proposed a machine learning based bug prediction approach for verification of SystemC designs. The proposed approach first collects data from repositories of SystemC projects, including product, process, and developer

Table 7.3: Summary of hybrid approaches for SystemC verification

References	**Techniques**	**Primary limitations**
Habibi and Tahar [38]	Model checking; simulation	Two-step translation and not automatic.
Satya [58]	Static analysis; dynamic POR	Strong restriction on SystemC designs; customized SystemC kernel.
Scoot [6]	Static analysis; model checking	Customized SystemC kernel.
VeriSTA [42, 43, 45]	Model checking; simulation	Large manual effort; limited support to SystemC features.
Aegis [1]	static analysis	False negative.
Ngo *et al.* [75–77]	Statistical model checking	Customized SystemC kernel; highly depends on the quality of generated stimuli.

metrics. With these collected data, a predictor model is trained. Finally, the trained model is used to predict whether or not a SystemC design is buggy. The authors conducted experiments on two open source SystemC projects, which show the high accuracy of the approach. However, this approach only predicts whether or not a design has bugs. It does not provide detailed information if a design is predicted as buggy, such as type and location of a bug. This compromises the usage and usefulness of the approach. Furthermore, the accuracy of the approach highly depends on the historic information of a project. If a project does not have detailed bug-fix information from revision logs, then the approach may not be able to identify whether or not there are bugs.

7.2 HARDWARE TROJAN DETECTION

Historically, most of computer security research was focused on software security. The underlying hardware was expected to be secure. However, hardware-related attacks have drawn attentions recently. Furthermore, many software security solutions rely on hardware-based root-of-trust that provides essential security functions. Thus, the assumption that the underlying hardware is secure is no longer the case. With the globalization of modern SoC design and manufacturing processes, and the emergence of new design paradigms such as outsourced design services and intensive usage of EDA tools, hardware vulnerabilities such as hardware Trojan attacks have raised serious concerns. As a result, a variety of hardware Trojan detection approaches have been developed recently.

So far, most hardware Trojan detection approaches are focused on RTL or lower level designs. There are a handful of Trojan detection approaches in RTL [2,3,93]. There are also various Trojan detection approaches that are focused on the gate level [9,50,55,79,84,103]. Post-silicon Trojan detection has also been studied [48, 72, 74, 78]. There has only been limited research on hardware Trojan detection for ESL designs. The pioneering work [97] detects hardware Trojans in behavioral SystemC designs using C++ control flow constructs and the property checker provided by a commercial high-level synthesis tool. The subsequent work [63] uses coverage-guided fuzz testing to detect hardware Trojans in behavioral SystemC designs. Both approaches are focused on high-level synthesizable SystemC designs which is a subset of ESL SystemC designs, while our approach presented in this dissertation is not restricted to the high-level synthesizable SystemC designs.

Chapter 8

CONCLUSIONS AND FUTURE RESEARCH

The growing complexity of modern SoCs and increasingly shortened time-to-market have pushed the design abstraction to the ESL to increase design productivity. SystemC is a widely used ESL modeling language in the semiconductor industry. ESL SystemC designs serve as executable specifications for the subsequent SoCs design flow. Therefore, undetected bugs in these designs may propagate to low-level implementations or even final silicon products. In this dissertation research, we have presented a framework to validate SystemC designs with automated test generation. This chapter concludes this dissertation and discusses some future research directions.

8.1 CONCLUSIONS

High-quality test cases are critical to simulation-based validation for SystemC designs. In this dissertation, we have presented a framework that includes multiple techniques for pre-silicon validation with SystemC designs. We have designed and developed multiple prototype tools with these techniques. We have also applied the framework to a couple of benchmarks to evaluate its effectiveness and efficiency. Specifically, the contributions of this dissertation are summarized as follows.

- Developed an approach to generating high-quality test cases for SystemC designs with symbolic execution techniques.

- Improved the scalability of test generation approach with binary-level concolic testing techniques and integrated ABV techniques to detect design errors.

- Developed an approach to detecting hardware Trojans in behavioral SystemC designs with selective concolic testing.

- Developed SCBench, a comprehensive suite of benchmark designs for SystemC verification and validation, which is freely available online.

Based on the experiments, our test generation approaches is able to generate high-quality test cases that achieve high code coverage and detect design errors effectively. During the experiments, our approaches detect two severe errors, one functional error and one out-of-bound access. We have applied our hardware Trojan detection approach to an open source SystemC benchmark with a variety of hardware Trojans. The experimental results demonstrate that the test cases generated by our approach are able to detect those hardware Trojans effectively and efficiently. Our extensive experiments show that our framework scales to designs with practical sizes as well.

8.2 FUTURE RESEARCH

This dissertation has presented a framework including multiple validation techniques such as symbolic execution and concolic testing for pre-silicon validation with SystemC designs. There are a few directions that may be explored based on this framework in the future.

First, our hardware Trojan detection approach may be used to detect hardware Trojans in Verilog RTL designs. The approach presented in this research is focused

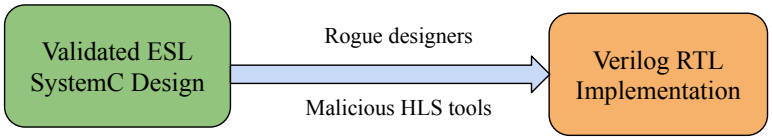

Figure 8.1: Adversarial threat model

on detecting hardware Trojans in ESL SystemC designs. After ESL SystemC designs are validated, they are translated into RTL implementations in Verilog or VHDL by design engineers manually or HLS tools automatically. Rogue designers or malicious HLS tools may insert hardware Trojans into RTL implementations, as shown in Figure 8.1. Therefore, Trojan free ESL SystemC designs become Trojan embedded RTL implementations. The inserted Trojans maybe propagate to lower level or even silicon products if they are not detected in the RTL implementations. Our current approach works on compiled SystemC binary executables. To leverage our approach, Verilog RTL implementations can be translated to cycle accurate SystemC designs first [1]. Then, our approach can be applied. With this use case, the validated and Trojan free ESL SystemC designs can be used as golden models for the Verilog RTL implementations. The workflow is demonstrated in Figure 8.2.

Second, fuzzing technique may be combined with our framework to improve the efficiency. Concolic testing requires constraint solving for each explored path, which is a time-consuming process. Fuzzing technique is very fast on the other hand. The high precision of concolic testing and high speed of fuzzing technique may be combined to improve efficiency of test generation for SystemC designs. Moreover, concolic testing is not suitable for computationally intensive SystemC designs, where fuzzing technique can play an important role.

[1]The open source tool Verilator can achieve this translation automatically. In addition, we target Verilog RTL implementations in this use case.

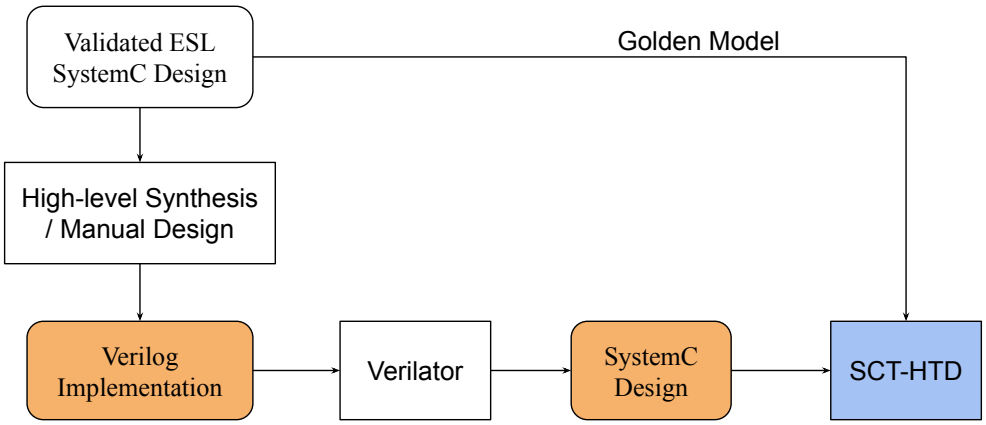

Figure 8.2: Workflow of hardware Trojan detection in Verilog RTL

Third, machine learning techniques may be integrated into the framework. On one hand, a new test case is selected at the beginning of each iteration of concolic test generation. Machine learning may be used to select the best test case in terms of certain criteria. On the other hand, many test cases may be generated for a SystemC design at the end of concolic testing process. It is time-consuming to use all the test cases for regression testing. Some test cases may be redundant in that they explore the same source code. Machine learning can be used to select a set of representative test cases for regression testing without losing effectiveness but increasing efficiency.

REFERENCES

[1] Aegis. `https://github.com/mglukhikh/aegis-systemc-benchmark`.

[2] A. Ahmed, F. Farahmandi, Y. Iskander, and P. Mishra. Scalable Hardware Trojan Activation by Interleaving Concrete Simulation and Symbolic Execution. In *Proceedings of International Test Conference (ITC)*, pages 1–10, Phoenix, AZ, USA, 2018.

[3] M. Banga and M. S. Hsiao. Trusted RTL: Trojan Detection Methodology in Pre-Silicon Designs. In *Proceedings of International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 56–59, Anaheim, CA, USA, 2010.

[4] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan. Hardware Trojan Attacks: Threat Analysis and Countermeasures. *Proceedings of the IEEE*, 102(8):1229–1247, 2014.

[5] B. Blanc, D. Kroening, and N. Sharygina. Scoot: A Tool for the Analysis of SystemC Models. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 467–470, Budapest, Hungary, 2008.

[6] N. Blanc and D. Kroening. Race Analysis for SystemC Using Model Checking. *ACM Transactions on Design Automation of Electronic Systems*, 15(3):21:1–21:32, 2010.

[7] F. Bruschi, F. Ferrandi, and D. Sciuto. A Framework for Functional Verification of SystemC Models. *International Journal of Parallel Programming*, 33(6):667–695, 2005.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, USA, 2008.

[9] B. Çakir and S. Malik. Hardware Trojan Detection for Gate-level ICs Using Signal Correlation Based Clustering. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 471–476, Grenoble, France, 2015.

[10] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara, and F. Xie. CRETE: A Versatile Binary-Level Concolic Testing Framework. In *Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE)*, Thessaloniki, Greece, 2018.

[11] M. Chen, P. Mishra, and D. Kalita. Towards RTL Test Generation from SystemC TLM Specifications. In *Proceedings of International High Level Design Validation and Test Workshop*, pages 91–96, Irvine, CA, USA, 2007.

[12] M. Chen, P. Mishra, and D. Kalita. Automatic RTL Test Generation from SystemC TLM Specifications. *ACM Transaction on Embedded Computing System*, 11(2):38:1–38:25, 2012.

[13] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 265–278, Newport Beach, CA, USA, 2011.

[14] C. Chou, C. Hsu, Y. Chao, and C. Huang. Formal Deadlock Checking on High-level SystemC Designs. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 794–799, San Jose, CA, USA, 2010.

[15] C.-N. Chou, C.-K. Chu, and C.-Y. Huang. Conquering the Scheduling Alternative Explosion Problem of SystemC Symbolic Simulation. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 685–690, San Jose, CA, USA, 2013.

[16] C.-N. Chou, Y.-S. Ho, C. Hsieh, and C.-Y. Huang. Symbolic Model Checking on SystemC Designs. In *Proceedings of Design Automation Conference (DAC)*, pages 327–333, San Francisco, CA, 2012.

[17] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. KRATOS: A Software Model Checker for SystemC. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, pages 310–316, Snowbird, UT, 2011.

[18] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri. Verifying SystemC: A Software Model Checking Approach. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD)*, pages 51–59, Lugano, Switzerland, 2010.

[19] A. Cimatti, I. Narasamdya, and M. Roveri. Boosting Lazy Abstraction for

SystemC with Partial Order Reduction. In *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 341–356, Saarbrücken, Germany, 2011.

[20] A. Cimatti, I. Narasamdya, and M. Roveri. Software Model Checking SystemC. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(5):774–787, 2013.

[21] Clang. `http://clang.llvm.org/docs/UsersManual.html`.

[22] E. Clarke, A. Biere, R. Raimiand, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[23] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[24] CLOC. `https://github.com/AlDanial/cloc`.

[25] K. Cong, F. Xie, and L. Lei. Automatic Concolic Test Generation with Virtual Prototypes for Post-silicon Validation. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 303–310, San Jose, CA, USA, 2013.

[26] K. Cong, F. Xie, and L. Lei. Symbolic Execution of Virtual Devices. In *Proceedings of International Conference on Quality Software (QSIC)*, pages 1–10, Nanjing, China, 2013.

[27] DOULOS. `www.doulos.com`.

[28] S. Dutta, D. Tabakov, and M. Y. Vardi. CHIMP: a Tool for Assertion-Based Dynamic Verification of SystemC Models. In *Proceedings of International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS)*, pages 38–45, Portland, OR, USA, 2013.

[29] S. Dutta and M. Y. Vardi. Assertion-Based Flow Monitoring of SystemC Models. In *Proceedings of International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 145–154, Lausanne, Switzerland, 2014.

[30] W. Ecker, V. Esen, T. Steininger, M. Velten, and M. Hull. Implementation of a Transaction Level Assertion Framework in SystemC. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 894–899, Nice, France, 2007.

[31] M. Efendioglu, A. Sen, and Y. Koroglu. Bug Prediction of SystemC Models Using Machine Learning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(3):419–429, 2019.

[32] F. Ferrandi, M. Rendine, and D. Sciuto. Functional Verification for SystemC Descriptions Using Constraint Solving. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 744–751, Paris, France, 2002.

[33] H. D. Foster. *Whitepaper: Trends in Functional Verification: A 2016 Industry Study.* Mentor Graphics.

[34] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[35] P. Godefroid and D. Pirottin. Refining Dependencies Improves Partial-Order Verification Methods. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, pages 438–449, Elounda, Greece, 1993.

[36] D. Große, H. M. Le, and R. Drechsler. Proving Transaction and System-level Properties of Untimed SystemC TLM Designs. In *Proceedings of International Conference Formal Methods and Models for Codesign (MEMOCODE)*, pages 113–122, Grenoble, France, 2010.

[37] G. D. Guglielmo, M. Fujita, F. Fummi, G. Pravadelli, and S. Soffia. EFSM-based Model-driven Approach to Concolic Testing of System-level Design. In *Proceedings of International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 201–209, Cambridge, UK, 2011.

[38] A. Habibi and S. Tahar. Design and Verification of SystemC Transaction-Level Models. *IEEE Transactions on Very Large Scale Integration Systems*, 14(1):57–68, 2006.

[39] R. Hajisheykhi, M. Roohitavaf, A. Ebnenasir, and S. Kulkarni. A Framework for Verification of SystemC TLM Programs with Model Slicing: A Case Study. In *Proceedings of Design Automation Conference (DAC)*, pages 1–6, Austin, TX, USA, 2016.

[40] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. CHStone: A benchmark program suite for practical C-based high-level synthesis. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1192–1195, Seattle, WA, USA, 2008.

[41] P. Herber, J. Fellmuth, and S. Glesner. Model Checking SystemC Designs

Using Timed Automata. In *Proceedings of International Conference on Hardware/Software Codesign and System Synthesis*, pages 131–136, Atlanta, GA, 2008.

[42] P. Herber, F. Friedemann, and S. Glesner. Combining Model Checking and Testing in a Continuous HW/SW Co-verification Process. In *Proceedings of International Conference on Tests and Proofs*, pages 121–136, Zurich, Switzerland, 2009.

[43] P. Herber and S. Glesner. A HW/SW Co-verification Framework for SystemC. *ACM Transactions on Embedded Computing Systems*, 12(1):61:1–61:23, 2013.

[44] P. Herber and B. Hünnemeyer. Formal Verification of SystemC Designs using the BLAST Software Model Checker. In *Proceedings of Workshop on Model-Based Architecting and Construction of Embedded Systems*, pages 44–53, Valencia, Spain, 2014.

[45] P. Herber, M. Pockrandt, and S. Glesner. Automated Conformance Evaluation of SystemC Designs using Timed Automata. In *Proceedings of European Test Symposium (ETS)*, pages 188–193, Praha, Czech Republic, 2010.

[46] V. Herdt, H. M. Le, and R. Drechsler. Verifying SystemC using Stateful Symbolic Simulation. In *Proceedings of Design Automation Conference (DAC)*, pages 1–6, San Francisco, CA, USA, 2015.

[47] V. Herdt, H. M. Le, D. Große, and R. Drechsler. Compiled Symbolic Simulation for SystemC. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, Austin, TX, USA, 2016.

[48] K. Hu, A. N. Nowroz, S. Reda, and F. Koushanfar. High-sensitivity Hardware Trojan Detection Using Multimodal Characterization. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1271–1276, Grenoble, France, 2013.

[49] IEEE Standards Association. *Standard SystemC Language Reference Manual*. IEEE Std. 1666-2011, 2011.

[50] D. Ismari, J. Plusquellic, C. Lamech, S. Bhunia, and F. Saqib. On Detecting Delay Anomalies Introduced by Hardware Trojans. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 1–7, Austin, TX, USA, 2016.

[51] A. D. Junior and D. J. Cecilio da Silva. Code-coverage Based Test Vector Generation for SystemC Designs. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 198–206, Porto Alegre, Brazil, 2007.

[52] M. Kallel, Y. Lahbib, R. Tourki, and A. Baganne. Verification of SystemC Transaction Level Models Using an Aspect-Oriented and Generic Approach. In *Proceedings of International Conference on Design Technology of Integrated Systems in Nanoscale Era*, pages 1–6, Hammamet, Tunisia, 2010.

[53] D. Karlsson, P. Eles, and Z. Peng. Formal Verification of SystemC Designs Using a Petri-Net Based Representation. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, Munich, Germany, 2006.

[54] James C King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[55] F. Koushanfar and A. Mirhoseini. A Unified Framework for Multimodal Submodular Integrated Circuits Trojan Detection. *IEEE Transaction on Information Forensics Security*, 6(1):162–174, 2011.

[56] D. Kroening and N. Sharygina. Formal Verification of SystemC by Automatic Hardware/Software Partitioning. In *Proceedings of International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 101–110, Verona, Italy, 2005.

[57] A. Kuehlmann and C. Eijk. Combinational and Sequential Equivalence Checking. In *Logic Synthesis and Verification*, pages 343–372. Springer, Boston, MA, 2002.

[58] S. Kundu, M. Ganai, and R. Gupta. Partial Order Reduction for Scalable Testing of SystemC TLM Designs. In *Proceedings of Design Automation Conference (DAC)*, pages 936–941, Anaheim, CA, USA, 2008.

[59] C. Kuznik and W Müller. Functional Coverage-driven Verification with SystemC on Multiple Level of Abstraction. In *Proceedings of Design and Verification Conference*, 2011.

[60] S.-T. Kwon. NoC(Network-on-Chip) Simulator. `http://opencores.org`.

[61] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO)*, pages 75–86, Washington, DC, USA, 2004.

[62] LCOV. `http://ltp.sourceforge.net/coverage/lcov/readme.php`.

[63] H. M. Le, D. Große, N. Bruns, and R. Drechsler. Detection of Hardware Trojans in SystemC HLS Designs via Coverage-guided Fuzzing. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 602–605, Florence, Italy, 2019.

[64] Hoang M Le, Daniel Große, Vladimir Herdt, and Rolf Drechsler. Verifying SystemC Using an Intermediate Verification Language and Symbolic Simulation. In *Proceedings of Design Automation Conference (DAC)*, pages 1–6, Austin, TX, 2013.

[65] T. Liebrenz, V. Klös, and P. Herber. Automatic Analysis and Abstraction for Model Checking HW/SW Co-Designs Modeled in SystemC. *ACM SIGAda Letters*, 36(2):9–17, 2016).

[66] B. Lin, J. Chen, and F. Xie. Selective Concolic Testing for Hardware Trojan Detection in Behavioral SystemC Designs. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 19–24, Grenoble, France, 2020.

[67] B. Lin and D. Qian. Regression Testing of Virtual Prototypes Using Symbolic Execution. *International Journal of Computer Science and Software Engineering*, 4(12):329–334, December 2015.

[68] B. Lin and F. Xie. SCBench: A Benchmark Design Suite for SystemC Verification and Validation. In *Prodeedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 440–445, Jeju, South Korean, 2018.

[69] B. Lin and F. Xie. A Systematic Investigation of State-of-the-Art SystemC Verification. *Journal of Circuits, Systems and Computers*, 29(15):1–27, 2020.

[70] B. Lin, Z. Yang, K. Cong, Z. Liao, T. Zhan, C. Havlicek, and F. Xie. Concolic Testing of SystemC Designs. In *Proceedings of International Symposium on Quality Electronic Design (ISQED)*, pages 1–7, Santa Clara, CA, USA, 2018.

[71] B. Lin, Z. Yang, K. Cong, and F. Xie. Generating High Coverage Tests for SystemC Designs Using Symbolic Execution. In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 166–171, Macau, China, 2016.

[72] E. Love, Y. Jin, and Y. Makris. Proof-Carrying Hardware Intellectual Property: A Pathway to Trusted Module Acquisition. *IEEE Transactions on Information Forensics and Security*, 7(1):25–40, Feb 2012.

[73] R. Milner. LCF: A Way of Doing Proofs with a Machine. In *Proceedings of International Symposium on Mathematical Foundations of Computer Science*, pages 146–159, Olomouc, Czechoslovakia, 1979.

[74] S. Narasimhan, D. Du, R. Chakraborty, S. Paul, F. Wolff, C. Papachristou, K. Roy, and S. Bhunia. Hardware Trojan Detection by Multiple-Parameter Side-Channel Analysis. *IEEE Transactions on Computers*, 62(11):2183–2195, 2013.

[75] V. C. Ngo and A. Legay. Formal Verification of Probabilistic SystemC Models with Statistical Model Checking. *Journal of Software: Evolution and Process*, 30(3):1–22, 2017.

[76] V. C. Ngo, A. Legay, and J. Quilbeuf. PSCV: A Runtime Verification Tool for Probabilistic SystemC Models. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, pages 84–91, Toronto, Canada, 2016.

[77] V. C. Ngo, A. Legay, and J. Quilbeuf. Statistical Model Checking for SystemC Models. In *Proceedings of International Symposium on High Assurance Systems Engineering (HASE)*, pages 197–204, Orlando, FL, USA, 2016.

[78] X. Ngo, I. Exurville, S. Bhasin, J. Danger, S. Guilley, Z. Najm, J. Rigaud, and B. Robisson. Hardware Trojan Detection by Delay and Electromagnetic Measurements. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 782–787, Grenoble, France, 2015.

[79] M. Oya, Y. Shi, M. Yanagisawa, and N. Togawa. A Score-based Classification Method for Identifying Hardware-trojans at Gate-level Netlists. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 465–470, Grenoble, France, 2015.

[80] L. Pierre and L. Ferro. A Tractable and Fast Method for Monitoring SystemC TLM Specifications. *IEEE Transactions on Computers*, 57(10):1346–1356, 2008.

[81] C. Pilato, K. Basu, F. Regazzoni, and R. Karri. Black-Hat High-Level Synthesis: Myth or Reality? *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(4):913–926, 2019.

[82] M. Pockrandt, P. Herber, and S. Glesner. Model Checking a SystemC/TLM Design of the AMBA AHB Protocol. In *Proceedings of Symposium on Embedded Systems for Real-Time Multimedia*, pages 66–75, Taipei, Taiwan, 2011.

[83] I. Polian, G. T. Becker, and F. Regazzoni. Trojans in Early Design Steps—An Emerging Threat. In *Proceedings of Conference on Trustworthy Manufacturing and Utilization of Secure Devices (TRUDEVICE)*, pages 55–60, Barcelona, Spain, 2016.

[84] J. Rajendran, V. Vedula, and R. Karri. Detecting Malicious Modifications of Data in Third-party Intellectual Property Cores. In *Proceedings of Design Automation Conference (DAC)*, pages 1–6, San Francisco, CA, USA, 2015.

[85] SCBench. `http://svl.cs.pdx.edu/scbench/scbench.html`.

[86] B.C. Schafer and A. Mahapatra. S2CBench: Synthesizable SystemC Benchmark Suite for High-Level Synthesis. *IEEE Embedded Systems Letters*, 6(3):53–56, September 2014.

[87] A. Sen. Concurrency-Oriented Verification and Coverage of System-Level Designs. *ACM Transactions on Design Automation of Electronic Systems*, 16(4):37:1–37:25, 2011.

[88] A. Sen and M. S. Abadir. Coverage Metrics for Verification of Concurrent SystemC Designs Using Mutation Testing. In *Proceedings of the High Level Design Validation and Test Workshop (HLDVT)*, pages 75–81, Anaheim, CA, 2010.

[89] A. Sen and O. Kalaci. Hybrid Dynamic Data Race Detection in SystemC. In *Proceedings of the Forum on Specification and Design Languages (FDL)*, pages 1–6, Munich, Germany, 2014.

[90] K. Sen. Concolic Testing. In *Proceedings of International Conference on*

Automated Software Engineering (ASE), pages 571–572, Atlanta, Georgia, USA, 2007.

[91] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of European Software Engineering Conference Held Jointly with International Symposium on Foundations of Software Engineering*, pages 263–272, Lisbon, Portugal, 2005.

[92] SEQ. `http://www.cprover.org/hardware/sequential-equivalence`.

[93] L. Shen, D. Mu, G. Cao, M. Qin, J. Blackstone, and R. Kastner. Symbolic Execution Based Test-patterns Generation Algorithm for Hardware Trojan Detection. *Computers & Security*, 78:267–280, 2018.

[94] J. Stoppe, R. Wille, and R. Drechsler. Cone of Influence Analysis at the Electronic System Level Using Machine Learning. In *Proceedings of Euromicro Conference on Digital System Design*, pages 582–587, Los Alamitos, CA, USA, 2013.

[95] SystemC Library. `http://www.accellera.org/downloads/standards/systemc`.

[96] M. Y. Vardi. Formal Techniques for SystemC Verification. In *Proceedings of Design Automation Conference (DAC)*, pages 188–192, San Diego, CA, USA, 2007.

[97] N. Veeranna and B. C. Schafer. Hardware Trojan Detection in Behavioral Intellectual Properties (IP's) Using Property Checking Techniques. *IEEE Transactions on Emerging Topics in Computing*, 5(4):576–585, 2017.

[98] N. Veeranna and B. C. Schafer. S3CBench: Synthesizable Security SystemC Benchmarks for High-Level Synthesis. *Journal of Hardware and Systems Security*, 1:103–113, 2017.

[99] L. Wang and M. S. Abadir. Data Mining in EDA - Basic Principles, Promises, and Constraints. In *Proceedings of Design Automation Conference (DAC)*, pages 1–6, San Francisco, CA, USA, 2014.

[100] L. Wang and M. Marek-Sadowska. Machine Learning in Simulation-Based Analysis. In *Proceedings of the Symposium on International Symposium on Physical Design (ISPD)*, pages 57–64, Monterey, CA, USA, 2015.

[101] M. Zalewski. Technical Whitepaper. `http://lcamtuf.coredump.cx/afl/technical_details.txt`.

[102] N. Zeng and W. Zhang. A Symbolic Partial Order Method for Verifying SystemC. In *Proceedings of Asia-Pacific Software Engineering Conference*, pages 271–278, Jeju, South Korea, 2014.

[103] J. Zhang, Feng Yuan, Lingxiao Wei, Zelong Sun, and Q. Xu. VeriTrust: Verification for Hardware Trust. In *Proceedings of Design Automation Conference (DAC)*, pages 1–8, Austin, TX, USA, 2013.